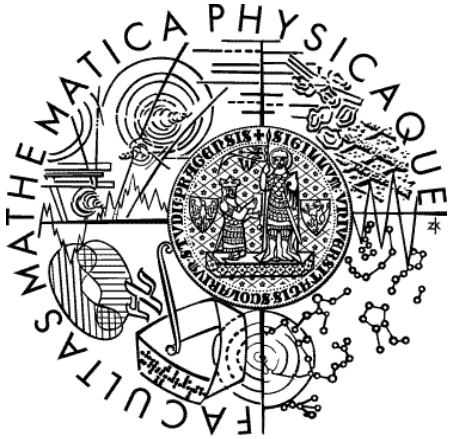
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKÁ PRÁCA



Martin Galajda

Algoritmy pro řešení hry Sokoban

Katedra teoretické informatiky a matematické logiky

Vedúci bakalárskej práce: RNDr. Pavel Surynek, Ph.D.

Študijný program:Informatika, programování

2009

Prehlasujem že som moju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním.

V Prahe dňa 7.8.2007 Martin Galajda

OBSAH

Názov práce: Algoritmy pro řešení hry Sokoban

Autor: Martin Galajda

Katedra: Katedra teoretické informatiky a matematické logiky

Vedúci bakalárskej práce: RNDr. Pavel Surynek, Ph.D.

E-mail vedúceho: Pavel.Surynek@mff.cuni.cz

Abstrakt:

Cieľom práce je priniesť uceleného sprievodcu problémom riešenia hry *SOKOBAN*. Práca sa zaoberá postupne problematikami definície problému, štúdiami jeho náročnosti, výberom vhodných algoritmov, a napokon aj ich implementáciou a testovaním. Štúdia demonštruje známe dôkazy NP-zložitosti a PSPACE-úplnosti problému. Neskôr sa práca venuje výberu a charakteristike vhodných algoritmov na riešenie problému. V rámci práce tiež vznikol program SokobanSolver, riešiaci množstvo bludísk hry SOKOBAN automaticky. Ukážeme si, ako sa tento program používa a ako funguje. Pozrieme sa tiež na program SokobanGenerator, ktorý vznikol ako podpora programu SokobanSolver. Tento program vytvára veľké množstvo náhodných bludísk, ktoré slúžia ako zdroj problémov programu SokobanSolver. Na konci práce sú uvedené údaje získané programom SokobanSolver. Bolo porovnané veľké množstvo riešení bludísk. Výsledky sú graficky spracované a diskutované. Najzaujímavejšie výsledky sú zhrnuté v závere celej práce.

Kľúčové slová : SOKOBAN, informované prehľadávanie, Astar, heuristika

Title: Algorithms for solving the Sokoban game

Author: Martin Galajda

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Pavel Surynek, Ph.D.

Supervisor's e-mail adress: Pavel.Surynek@mff.cuni.cz

Abstract:

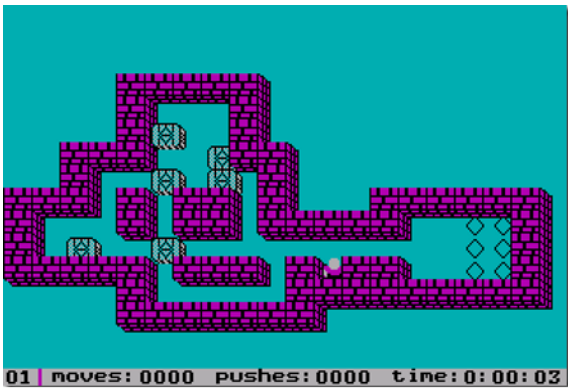
The task of this work is to bring a complete guide for solving a SOKOBAN game problem. The work follow subsequently issues of studies of problem complexity, its definition, selection of appropriate algorithms, as well as their implementation and testing. The study demonstrate

the manifest, that problem belongs to a NP-hard set of problems as well as PSPACE-complete set of problems. Fundamental and also advanced techniques of state space searching are studied as means for solving defined problem, memory and time characteristics of these techniques are presented and derived. In order to gain statistical information about efficiency of demonstrated algorithms and heuristics, an aplication SokobanSolver was built. This program can solve many SOKOBAN problems automatically. We described principles of functional parts and main data structures of this program and also we discuss program *SokobanGenerator,* aplication that was built to supply SokobanSolver with many random built SOKOBAN problems. Techniques explained are tested by lately named programs, and results are discussed at the very end of the paper.

Keywords: SOKOBAN, informed search, Astar, heuristics

1 Úvod

SOKOBAN je logická hra, ktorá vznikla v osemdesiatych rokoch minulého storočia v Japonsku. Jej Autorom je Hiroyuki Imabayashi[1]. Pôvodná hra simuluje prácu skladníka (skladník je preklad slova „sokoban“ z japončiny). Úlohou hráča je prostredníctvom skladníka premiestniť všetky krabice z počiatočných pozícií na označené cieľové pozície. Pritom hráč nemá možnosť premiestniť viac než jednu krabicu súčasne. Navyše hráč nemá možnosť krabice ťahať, môže ich iba tlačiť[2]. V priebehu rokov vzniklo veľké množstvo variácií tejto úspešnej hry s rôznymi obmenami pravidiel. My sa v práci budeme zaoberať pôvodným návrhom.



Obrázok 1: Pôvodná verzia hry SOKOBAN

Napriek spomenutým jednoduchým pravidlám má táto hra zaujímavé pamäťové a priestorové nároky. Preto je vďačným objektom skúmania algoritmov pre automatické vyhľadávania riešenia. Konkrétne si ukážeme, že hra *SOKOBAN* je *PSPACE*-úplná[3] a jej vyriešenie je dokonca *NP*-ťažké[8]. Zložitosť hry je daná nielen rýchlosťou vetvenia kde každým prechodom od stavu hry do ďalšieho stavu hry dochádza v najhoršom prípade až k zoštvornásobeniu počtu možností hry, ale aj takzvanou hĺbkou problému. Hĺbka problému vyjadruje počet krokov potrebných pre nájdenie riešenia. Niektoré úrovne hry potrebujú na vyriešenie niekoľko stoviek, dokonca tisícov krokov. Pri takto definovaných problémoch sa programy ale aj ľudia spoliehajú na rôzne heuristiky, ktoré súvisia so znalosťou problému. O týchto heuristikách budeme písať v nasledujúcich kapitolách a taktiež aj o algoritmoch ktoré

dokážu tieto heuristiky využívať.

SokobanSolver, program, ktorý vznikol v rámci tejto práce, je založený na pôvodnom návrhu hry z osemdesiatych rokov. Program dokáže riešiť bludisko rôznymi algoritmami, ktoré kombinuje s viacerými heuristikami a dokáže pracovať na viacerých typoch dátových štruktúr. Výstupom programu je súbor s hodnotami času potrebného na vyriešenie bludiska a počtu krokov, potrebných na nájdenie riešenia. Tieto hodnoty sú v tabuľke priradené konkrétnemu bludisku a použitému algoritmu.

Spolu s programom *SokobanSolver* určenému na riešenie SOKOBAN problémov, vznikol program SokobanGenerator, ktorý umožňuje pohodlne vytvárať veľké množstvá náhodných bludísk. Program sa snaží naimplementovaným algoritmom[14] generovať čo najväčší počet zmysluplných a riešiteľných bludísk, no ako tvrdia autori použitého algoritmu: „jediný spôsob ktorým dokážete riešiteľnosť bludiska je, že ho vyriešite“ a preto napriek postupu ktorým sa bludiská generujú, nemožno zaručiť riešiteľnosť každého z nich. Počet riešiteľných bludísk na výstupe programu SokobanGenerator je však pre naše použitie uspokojivý a preto vhodný ako zdroj pre testovaný program SokobanSolver.

Cieľom tejto práce, je priniesť komplexný pohľad na súčasné postupy riešenia hry SOKOBAN ale aj iných podobných problémov. Ukázať základné princípy ich fungovania a demonštrovať teoretické znalosti o výkonnosti používaných algoritmov. Následne sa pozrieme aj na to, ako sú predvádzané algoritmy implementované v programe SokobanSolver. Ten preskúšame v záverečnom teste, aby sme mohli porovnať teoretické predpoklady o výkonnosti použitých algoritmov so skutočnými výsledkami.

2 Definícia problému

V počiatkoch všetkých úvah zaoberajúcich sa riešením problému musí stáť pevná definícia tohto problému. O čosi také sme sa pokúsili už v úvode, popísaním pravidiel hry SOKOBAN. No ak sa chceme hlbšie venovať štúdiu problému, nemôžeme sa s takouto definíciou uspokojiť. Pokúsime sa teda definovať problém inak. K tomuto účelu slúži špeciálna trieda popisovacích jazykov. Do tejto triedy patrí aj jazyk nazvaný STRIPS. Je to práve ten jazyk, ktorým sa my pokúsime popísať problém hry SOKOBAN.

Syntax a sémantika tohto jazyka nie je zložitá, no napriek tomu siaha nad rámec tejto práce a z toho dôvodu ju tu neuverejním. V prípade, že čitateľ bude mať záujem sa o tomto jazyku dozvedieť viac, alebo bude mať nejasnosti ohľadom definície problému, odporúčam preštudovať si čo o jazyku STRIPS hovorí publikácia Artificial Intelligence a Modern Approach[10]. Ďaľšou možnosťou je prednáška pána Küngasa z Nórska[11]. Oba tieto materiály boli použité ako zdrojové podklady pre túto kapitolu.

V nasledujúcich riadkoch uvediem priamo kód v STRIPS jazyku. V príklade sa obmedzím na problém s dvoma krabicami, a dvoma stenami. Viac krabíc sa do kódu zapíše analogicky k uvedeným krabiciam a stenám. Pre nás je dôležité, že týmto obmedzením sa nezmenia definície funkcií.

Init (Position (R, 1, 1) ∧ Position (B1, 2, 2) ∧ Position(B2, 3, 2) ∧ Position (W1, 1, 2)

∧Position (W2, 1, 3) ∧ Position (FREE1, 2, 1) ∧ Position(FREE2, 3, 1)

∧Position (FREE3, 2, 3) ∧ Position(FREE4, 3, 3) ∧ Robot (R) ∧ Box (B1)

∧ Box (B2) ∧ Wall (W1) ∧ Wall (W2) ∧ Free(FREE1)

∧ Free(FREE2) ∧ Free(FREE3) ∧ Free(FREE4))

V tomto kroku sme definovali úvodný stav, tj. polohu všetkých objektov na hracom pláne a ich príslušnosť do tried. Výrazy v Position, Box, Finall a Wall sú funkcie, výrazy v zátvorkách sú konštanty.

Goal(Position (B1,2,3)∧Position(B2,3,3))

Goal(Position(B1,3,3)∧Position(B2,2,3))

V tomto kroku sme definovali ako má vyzerať cieľový stav. V našom prípade to znamená, že obe krabice musia byť na svojich cieľových pozíciách.

Action(MoveLeft(x,y,r,f),

PRECOND: Position(r,y,x)∧Position(f,y,x-1)∧Robot(r)∧Free(f)

EFECT: Position(r,y,x-1)∧¬ Position(r,y,x)∧Position(f,y,x)∧¬ Position(f,y,x-1)

)

Definovali sme pohyb vľavo. Výraz PRECOND označuje stav, v ktorom sa popisovaný problém musí nachádzať aby sa vykonala funkcia MoveLeft. Význam podmienky je nasledujúci. Musí byť definovaná pozícia [y,x] na ktorej je objekt r. Podmienka Robot(r) vynucuje aby premenná r bol robot. Ďalšia podmienka vyžaduje aby bola definovaná pozícia[y,x-1] pre objekt f, pre ktorý musí byť definované Free(f), čo znamená, že to musí byť voľné pole. Ak sa podmienkam vyhovie, vykoná sa EFECT. To znamená, že sa vytvorí nový záznam o pozícii so súradnicami [y,x-1] a objektom robot. Ďalší výraz vymaže pôvodný záznam. Presunuli sme robota doľava. Zvyšok výrazu vytvorí nový záznam o voľnom mieste na pôvodnej pozícii robota. Starý záznam vymaže. Zvyšné funkcie pohybu sa chovajú analogicky.

Action(MoveRight(x,y,r,f),

PRECOND: Position(r,y,x)∧Position(f,y,x+1)∧Robot(r)∧Free(f)

EFECT: Position(r,y,x+1)∧¬ Position(r,y,x)∧Position(f,y,x)∧¬ Position(f,y,x+1)

)

Action(MoveUp(x,y,r,f),

PRECOND: Position(r,y,x)∧Position(f,y-1,x)∧Robot(r)∧Free(f)

EFECT: Position(r,y-1,x)∧¬ Position(r,y,x)∧Position(f,y,x)∧¬ Position(f,y-1,x)

)

Action(MoveDown(x,y,r,f),

PRECOND: Position(r,y,x)∧Position(f,y+1,x)∧Robot(r)∧Free(f)

EFECT: Position(r,y+1,x)∧¬ Position(r,y,x)∧Position(f,y,x)∧¬ Position(f,y+1,x)

)

Action (PushLeft(x,y,r,f,b),

PRECOND: Position(r,y,x)∧Position(b,y,x-1)∧Position(f,y,x-2)∧Robot(r)∧Box(b)

∧Free(f)

EFECT: Position(r,y,x-1)∧¬ Position(r,y,x)∧Position(b,y,x-2)∧¬ Position(b,y,x-1)

Position(f,y,x)∧¬ Position(f,y,x-2)

)

Definovali sme posun krabice doľava. Význam podmienky je nasledujúci. Musí existovať záznam o pozícii objektu r na súradniciach [y,x] a navyše musí existovať záznam Robot(r). Inými slovami, musí existovať robot na súradniciach [y,x]. Ďalej musí existovať záznam o krabici na pozícii [y,x-1], teda na pozícii vľavo od hráča. Nakoniec musí existovať voľná plocha na pozícii[y,x-2], teda na mieste, kam sa krabica chystá posunúť. V prípade úspechu, sa vytvorí nový záznam o robotovi na pozícii o jedna vľavo. Starý záznam sa vymaže. Vytvorí sa nový záznam o krabici naľavo od starého. Starý sa vymaže. Voľná pozícia sa „presunie“ na pôvodnú pozíciu hráča. Pohyb je hotový. Ostatné prípady posuvu sú analogické.

Action(PushRight(x,y,r,f,b),

PRECOND: Position(r,y,x)∧Position(b,y,x+1)∧Position(f,y,x+2)∧Robot(r)∧Box(b)

∧Free(f)

EFECT: Position(r,y,x+1)∧¬ Position(r,y,x)∧Position(b,y,x+2)∧¬ Position(b,y,x+1)

Position(f,y,x)∧¬ Position(f,y,x+2)

)

Action(PushUp(x,y,r,f,b),

PRECOND: Position(r,y,x)∧Position(b,y-1,x)∧Position(f,y-2,x)∧Robot(r)∧Box(b)

∧Free(f)

EFECT: Position(r,y-1,x)∧¬ Position(r,y,x)∧Position(b,y-2,x)∧¬ Position(b,y-1,x)

Position(f,y,x)∧¬ Position(f,y-2,x)

)

Action(PushDown(x,y,r,f,b),

PRECOND: Position(r,y,x)∧Position(b,y+1,x)∧Position(f,y+2,x)∧Robot(r)∧Box(b)

∧Free(f)

EFECT: Position(r,y+1,x)∧¬ Position(r,y,x)∧Position(b,y+2,x)∧¬ Position(b,y+1,x)

Position(f,y,x)∧¬ Position(f,y+2,x)

)

Týmto sme ukončili formálny popis problému v jazyku STRIPS.

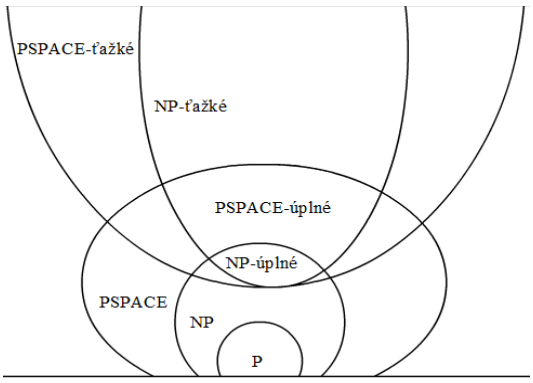
Teraz, keď už máme hotovú definíciu problému, môžeme sa pustiť do hlbšieho skúmania problému. V nasledujúcich kapitolách si teda povieme niečo o zložitosti nášho problému.

3 PSPACE úplnosť

V úvode práce sa odvolávam na tvrdenie pána Culbersona z univerzity v Alberte, ktorý tvrdí, že Sokoban je PSPACE-úplný problém. Toto tvrdenie je zverejnené v článku s názvom „Sokoban is PSPACE-complete“[3] kde je taktiež dokázané. V tejto kapitole sa na spomínaný dôkaz pozrieme bližšie.

Triedu PSPACE-úplných problémov najčastejšie definujeme pomocou inej triedy zložitosti a to pomocou triedy PSPACE. Definícia problémov patriacich do PSPACE je podľa dostupnej literatúry [4] nasledovná: “Povedzme, že rozhodovacia úloha ϑ je v triede PSPACE, ak existuje (deterministický) algoritmus, ktorý rieši ϑ a ktorý má pamäťovú zložitosť najhoršieho prípadu O(p(n)) pre nejaký polynóm.“ S touto znalosťou už môžeme definovať PSPACE-úplnú triedu problémov. PSPACE-úplným problémom nazveme problém, ktorý je PSPACE a akýkoľvek problém z tejto množiny môže byť na neho redukovaný v polynomickom čase (algoritmom pracujúcim s polynomickou časovou zložitosťou O(p(n)))[5]. Z definície redukcie problémov [4] vyplýva, že na PSPACE-úplné problémy sa možno pozerať ako na najťažšie problémy z množiny PSPACE. Z uvedenej definície PSPACE-úplnych problémov budeme vychádzať aj v nasledujúcom dôkaze.

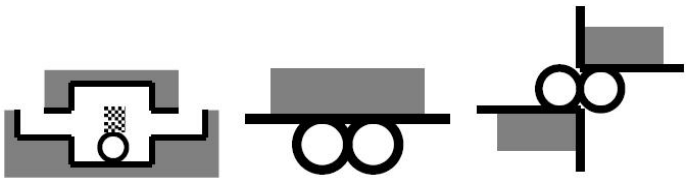
Zložením definícií rôznych zložitostných problémov[4][5] je možné získať náhľad na celkový zložitostný priestor. Ako však aj autori týchto definícií varujú, mnoho sa toho o rozdelení zložitostí ešte nepodarilo dokázať. Preto je tento náhľad, ktorý uvádzam na obrázku 2, len jednou z možností, ako by mohol byť tento priestor rozdelený.



Obrázok 2: Jedno z možných rozdelení zložitostného priestoru, vyjdrené Eulerovými grafmi.

Ukážeme si, ako je možné z nekonečnej verzie *SOKOBAN* problému, kde iba konečný počet krabíc je mimo svoju cieľovú polohu, emulovať Turingov stroj v lineárnom čase. Následným obmedzením pásky na konečnú dĺžku ukážeme, že konečne zadaný problém *SOKOBAN* je PSPACE-ťažký[6]. Nakoľko konečne zadaný problém je taktiež aj z množiny PSPACE[7] platí, že problém musí byť z množiny PSPACE úplných problémov[3].

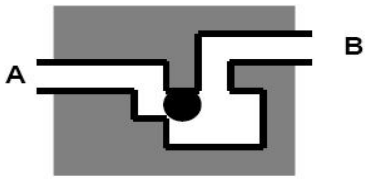
V úvode dôkazu sa pozrieme na postavenia krabíc a stien z prostredia hry SKOBAN, ktoré budú mať vďaka svojím špeciálnym vlastnostiam v dôkaze kľúčovú úlohu. Tieto postavenia sú náchylné na nezotaviteľnosť. To znamená, že z týchto stavov je možné sa dostať do situácie z ktorej neexistuje postupnosť krokov taká, že na konci tejto postupnosti je stav so všetkými krabicami na finálnych pozíciach. Niektoré nezotaviteľné pozície, ktorých vlastnosti budeme aj my neskôr využívať máme na obrázku 3.



Obrázok 3: Nezotaviteľné stavy. V prvom prípade je možné posunúť krabicu len do nezotaviteľnej pozície. V druhom a treťom prípade nemožno krabicami pohnúť vôbec.

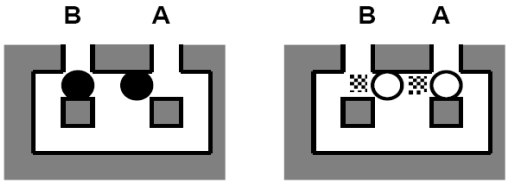
Kombinovaním týchto náchylných stavov dostaneme takzvané skrinky(devices). Pomocou týchto skriniek s vhodnou konfiguráciou náchylných stavov môžeme

kontrolovať tok programu tak, že nútime hráča správať sa tak, ako nám to vyhovuje tým, že obmedzíme možnosti jeho rozhodovania. Prvou uvažovanou skrinkou je jednosmerná skrinka(*One-Way device)*. Ako naznačuje jej názov, jedná sa o skrinku ktorá bude riadiť dopravu v bludisku tým, že dovoľuje prechod len jedným smerom. Skrinka ktorú máme na obrázku dovoľuje iba prechod zo vstupu A do výstupu B, a to kedykoľvek keď na ňu narazíme. Príchodom zo vstupu B skrinku zablokujeme.



Obrázok 4: Jednosmerná skrinka.

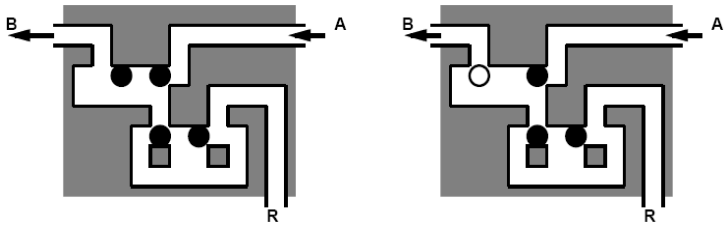
Ďalšou skrinkou je obracač*(Reverser)*. Má dva základné stavy. Prvý, na ľavom obrázku, je vo vyriešenej pozícii. Druhý, na pravom obrázku, je v pozícii ktorá sa objaví potom, ak opustíme skrinku cez otvor B. Táto skrinka umožňuje hráčovi prejsť z otvoru A do otvoru B a opustiť zariadenie. Všimnime si ale zaujímavú vlastnosť, a tou je, že kým sa hráč nevráti zasa otvorom B a neopustí skrinku otvorom A, krabice v skrinke nebudú vo vyriešenej pozícii. Táto skrinka sa teda využíva tam kde potrebujeme docieliť, aby sa hráč vrátil cestou ktorou prišiel.



Obrázok 5: Obracač. Prechod z A do B implikuje spiatočný prechod z B do A.

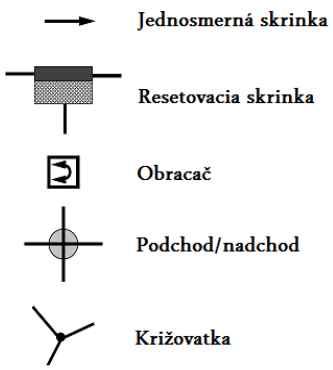
Poslednou a najkomplikovanejšou skrinkou je resetovacia skrinka*(Pass-Reset)*. Skrinku máme na obrázku 6. Šípky naznačujú, že na otvoroch A a B sú nainštalované jednosmerné skrinky. Na prvom obrázku je skrinka v takzvanom uzavretom stave, krabice sú na svojich miestach. Tento stav neumožňuje prechod zariadením, vstupom do otvoru A, a vystúpením z otvoru B, a zároveň zanechaním zariadenia v riešiteľnej pozícii. Jediným možným vstupom neporušujúcim riešiteľnosť je vstup R. Vstupom do R nastavíme skrinku do stavu v akom sa nachádza na obrázku vpravo. Všimnime si, že

na vstupe R je nainštalovaný obracač, čo znamená, že po nastavení skrinky musíme zariadenie opustiť opäť otvorom R. V ďalšom kroku už môžeme do zariadenia vstúpiť vchodom A a opustiť východom B, zanechajúc tak zariadenie opäť vo vyriešenom stave aký vidíme na ľavom obrázku (resetujeme zariadenie prechodom). Zariadenie bude uzavreté a pred ďalším prechodom ho musíme opäť nastaviť do pozície, ako je naznačené na pravom obrázku. Na záver si ešte overme, že hráč nemôže vstúpiť vchodom R a vystúpiť východom B. V prípade, ak hráč vstúpi vchodom R a opustí zariadenie otvorom B, obracač bude v nevyriešenom stave a neumožní opätovný vstup do zariadenia. Vstup do otvoru B je zakázaný jednosmerným zariadením. Keďže zariadenie bolo opustené otvorom B, krabica vľavo hore je vo vyriešenom stave, čo bráni použitiu vstupu A. Nemožno teda použiť ani vstup A, ani vstup B, ani vstup R, a zariadenie ostalo v nevyriešenej pozícii.



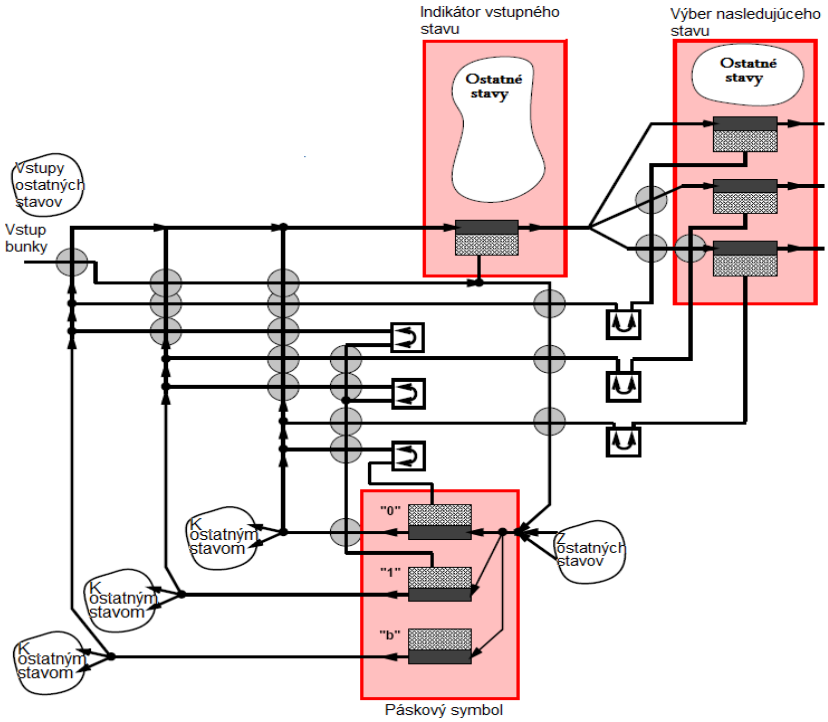
Obrázok 6: Resetovacia skrinka. Pred prechodom skrinkou zo vstupu A na výstup B je nutné najprv upraviť vstupom R skrinku do pozície napravo. Prechodom sa skrinka opäť resetuje.

V ďalšej časti si povieme ako zostaviť Turingov stroj s pomocou opísaných troch zariadení a dvoch servisných zariadení reprezentujúcich jednoduchú križovatku *(Junction)* a štruktúru, ktorá funguje ako trojrozmerná križovatka *(Crossover)*, teda akýsi nadchod/podchod. Zoznam všetkých použitých objektov je na obrázku 7.



Obrázok 7: Schematické symboly zariadení použitých pri konštrukcii Turingovho stroja

Na popísanie Turingovho stroja použijeme schému v ktorej budú jednotlivé zariadenia nahradené svojimi schematickými značkami, ako sme ich uviedli na predchádzajúcom obrázku. Základnú formu máme na obrázku 8. Štruktúra na obrázku predstavuje časť Turingovho stroja, zodpovednú za rozhodnutie o novom stave stroja a posune na ďalšiu bunku pásky na základe aktuálneho symbolu na páske a aktuálneho stavu. Táto štruktúra zobrazuje iba zariadenia jedného stavu stroja. Ostatné stavy potrebujú kópiu zariadenia uvedeného na obrázku, okrem časti „Páskový symbol“, tá je zdieľaná všetkými stavmi.



Obrázok 8: Schéma bunky Turingovho stroja, zobrazuje jeden páskový symbol s jedným stavom stroja, spolu s logikou rozhodovania stroja na základe ich hodnôt.

Na vstupe bude jedinou otvorenou resetovacou skrinkou niektorá z tých, čo sa

nachádzajú v regióne „Páskový symbol“ a bude to práve tá, ktorá reprezentuje aktuálne prečítaný symbol na páske. V našom prípade „1“, „0“ alebo „b“ ako symbol pre prázdnu pásku.

Predstavme si, že Turingov stroj prečíta symbol X na páske a momentálne sa nachádza v stave Y. V našom prípade to znamená, že sme vstúpili do inštancie bunky z obrázka 8(rozšírenej o schémy ďalších stavov), ktorá má v regióne Páskový symbol resetovaciu skrinku otvorenú pre symbol X. Do bunky vstúpime cestou, ktorá v regióne indikátoru vstupného stavu otvorí resetovaciu skrinku reprezentujúcu stav Y. V prípade iného stavu by sme vošli do tej istej inštancie, avšak inou cestou, čím by sme otvorili inú resetovaciu skrinku. Následne nás cesta zavedie do regiónu páskového symbolu. V tomto regióne budeme nasmerovaný tak, aby sme otvorili resetovaciu skrinku v regióne výberu nasledujúceho stavu. Cestou uzavrieme resetovaciu skrinku v regióne páskového symbolu, ktorou sme prešli. Výber resetovacej skrinky ďalšieho stavu je ovplyvnený symbolom X z pásky a symbolom Y reprezentujúcim stav stroja. Takto je zaistená emulácia práce Turingovho stroja, prechodom hráča jednotlivými bunkami, kde každý symbol pásky má vlastnú inštanciu bunky. Aby sme však mohli vysloviť tvrdenie, že riešením SOKOBAN bludiska, je možné simulovať prácu Turingovho stroja, musíme sa zamyslieť ešte nad niekoľkými otázkami:

1. Ak Turingov stroj zastane a prijme, v stroji sa môže vyskytovať ľubovolný počet nenavštívených buniek pásky. V prenesenom význame to znamená, že sa v nenavštívených bunkách budú nachádzať zariadenia v otvorenom stave, čo znamená že *SOKOBAN* ešte nebol vyriešený.
2. Kým niektoré bunky na páske obsahujú znaky, ostatné za koncom vstupu môžu byť prázdne. Ak ich tam bude príliš veľa nepodarí sa nám splniť podmienku lineárnosti.
3. Turingov stroj môže navštíviť každú bunku na konečnej páske, napriek tomu však skončiť bez prijatia, alebo neskončiť v dôsledku zacyklenia. V jednom či druhom prípade, problém môže byť napriek tomu vyriešený, pretože hráčovi sa nemusí podariť resetovať *páskový symbol* pred opustením bunky.

Aby sme vyriešili prvý problém, pridáme takzvanú *Zastav-Prijmi chodbu(Halt-Accept corridor)*, do ktorej je možné sa dostať iba vystúpením z bunky Turingovho stroja

emulovaním prechodu do prijímacieho konečného stavu. Z tejto chodby, budú viesť spojenia do každého regiónu páskového symbolu, a kontrolované resetovacími skrinkami v regiónoch indikátorov vstupných stavov budú umožňovať uzavrieť všetky otvorené resetovacie skrinky v regiónoch páskových symbolov zariadenia.

Aby sme zaručili emuláciu v lineárnom čase, pridáme na pásku špeciálny koncový *(end-of-tape)* symbol, ktorý sa bude správať presne tak, ako prázdny symbol, až na to, že umožní hráčovi resetovať páskový koncový symbol v nasledujúcom regióne páskového symbolu. Všetky regióny páskového symbolu za koncovým symbolom tak budú implicitne uzavreté.

Aby sme zabránili falošným riešeniam, tak zastav-prijmi chodba bude mať implicitne jednu krabicu mimo cieľového stavu. *SOKOBAN* bude takto môcť byť vyriešený jedine ak Turingov stroj zastane v prijímacom stave, čo umožní hráčovi dostať na miesto túto poslednú krabicu.

Ak týmto spôsobom upravíme naše riešenie, výsledkom bude, že *SOKOBAN* problém bude mať riešenie vtedy a len vtedy, ak emulovaný Turingov stroj zastane v prijímacom stave. To je to, čo bolo treba dokázať.

4 NP-ťažkosť

Okrem pamäťovej náročnosti sa podarilo o skupine hier, do ktorých patrí aj SOKOBAN dokázať, že problémy patriace do tejto skupiny sú NP-ťažké. Toto tvrdenie vyslovili a dokázali Demaine D. a Hoffman M. v práci „Pushing Blocks is NP-Complete for Non crossing Solution Paths“[8]. V tejto kapitole sa budeme venovať práve tomuto tvrdeniu a ukážeme si aj, ako páni Demain s Hoffmanom konštruovali dôkaz zaručujúci pravdivosť ich vlastného tvrdenia.

Na úvod je však nutné definovať pojem NP-zložitosti, pretože táto definícia je kľúčová pri konštrukcii dôkazu pôvodného tvrdenia. Problém H, je NP-ťažkým problémom, vtedy a len vtedy, ak existuje NP-úplný problém L taký, ktorý je Turingovo reducibilný (prevediteľný v polynomickom čase Turingovým strojom) na problém H. O probléme H pritom platí, že sám sa nemusí nachádzať v skupine NP-úplných problémov[4]. Inými slovami, problém H je NP-ťažký, ak je aspoň taký ťažký ako ľubovolný problém zo skupiny NP-úplných problémov. Pre lepšiu predstavu odporúčam sa vrátiť k obrázku 2, kde sú pomocou množín ilustrované jednotlivé vzťahy zložitostného priestoru. Teraz máme dostatočné vedomosti o NP-zložitých problémoch na to, aby sme sa mohli pozrieť na zmieňovaný dôkaz.

Podľa autorov dokazovaného tvrdenia je možné rôzne problémy tlačenia prekážok rozdeliť podľa základných charakteristík do niekoľkých skupín. Sú nimi napríklad skupiny *PUSH-PUSH*, ktorou sú označované problémy, kde sa prekážky musia posúvať o maximálnu možnú dĺžku ktorú im povoľuje bludisko, či problémy *PUSH-X*, kde patria problémy pri ktorých hráč nesmie križovať svoju vlastnú cestu. Nás však bude zaujímať tretia skupina problémov, a to *PUSH-k*, kde *k* označuje koľko prekážok je hráč schopný pred sebou tlačiť. Dôkaz PUSH-*k* pre ľubovolné celé *k* je rozšírením dôkazu PUSH-1 a nebudeme ho v tejto práci uvádzať. Pozrieme sa iba na tú časť dôkazu, ktorá končí dokázaním tvrdenia, že PUSH-1 je NP-ťažké.

Problém, ktorý sa chystáme previesť, je 3-ofarbenie rovinného grafu. O tomto probléme je známe, že je NP-úplný[9]. Ak sa nám podarí v polynomicky dlhom čase transformovať problém ofarbenia daného grafu na problém vyriešenia SOKOBAN bludiska, a ešte sa nám navyše podarí dokázať, že tento novovzniknutý problém je

riešiteľný práve vtedy a len vtedy ak pôvodný graf je 3-ofarbiteľný, tak budeme môcť prehlásiť, že SOKOBAN(ako aj všetky PUSH-1 problémy) je NP-ťažký problém. Toto tvrdenie vyplýva priamo z definície NP-ťažkosti.

Majme teda zadaný nejaký graf *G*, o ktorom platí, že je neorientovaný a rovinný. Ďalej máme graf $\vec{G}$, ktorý vznikol z grafu *G* nahradením každej neorientovanej hrany dvojicou opačne orientovaných hrán. V takomto grafe $\vec{G}$, existuje Eulerova kružnica. Dôkaz tohto tvrdenia sa môže opierať o fakt, že každý vrchol tohto grafu má rovnaký počet vstupujúcich a vystupujúcich hrán. Dokonca môžeme predpokladať, že v každom takomto grafe existuje rovinná Eulerova kružnica *T[8]*.

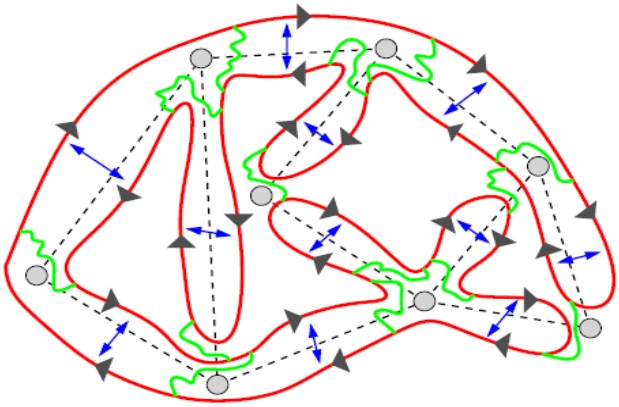
Kružnica *T* prechádza všetkými bodmi $\vec{G}$. Použijeme ju preto na prechod grafu $\vec{G}$ pri jeho ofarbovaní. Analógiou prechodu grafu pri ofarbovaní je prechod špeciálne vytvoreného bludiska SOKOBAN hráčom. Na to, ako toto bludisko vytvoríme sa pozrieme však až neskôr.

Pri ofarbovaní grafu postupujeme nasledovne. Pri každom odchode z vrcholu vyberieme tomuto vrcholu farbu a vždy keď budeme prechádzať druhou hranou z tých dvoch, ktoré tvoria hranu v pôvodnom *G,* skontrolujeme farby susedných vrcholov. Na tieto úlohy sú však kvôli podmienke korektného ofarbenia kladené isté nároky. Podľa týchto podmienok musíme úlohy patrične spresniť.

Jednou podmienkou je konzistencia. To znamená, že musíme zaručiť, že pri viacnásobnom opúšťaní vrcholu nebudeme meniť jeho pôvodnú farbu, ale stále ho ofarbíme rovnako. Druhou podmienkou je, že každé dva susedné vrcholy musia dostať rozdielnu farbu.

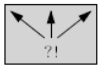
Aby sme zaistili tieto dve podmienky, doplníme pôvodný graf, o takzvané skrinky*(gadgets)*. Aby sme teda zaistili konzistenciu farieb, necháme každé dve hrany vystupujúce z jedného vrcholu vstúpiť do takzvanej skrinky konzistencie (*consistency gadget*). V grafe si túto úpravu označíme vlnitou čiarou. Samozrejme sa nám nemusí podariť spojiť každé dve hrany navzájom bez toho, aby sme neporušili rovinnosť. To však ani nie je nutné vzhľadom k tomu, že ak vyrobíme aspoň reťaz z hrán pospájaných skrinkami konzistencie o rovnakú farbu sa za nás postará tranzitivita. O druhú podmienku sa bude starať farbiaca križovatka*(coloring junction)*. Táto skrinka bráni tomu, aby mali susedné vrcholy rovnakú farbu. V grafe na obrázku, je farbiaca

križovatka označená obojstrannou šípkou.



Obrázok 9: Graf $\vec{G}$ s naznačenou kružnicou a skrinkami. Červená čiara označuje Eulerovu kružnicu, modré šípky označujú farbiace križovatky, a zelené čiary skrinky konzistencie.

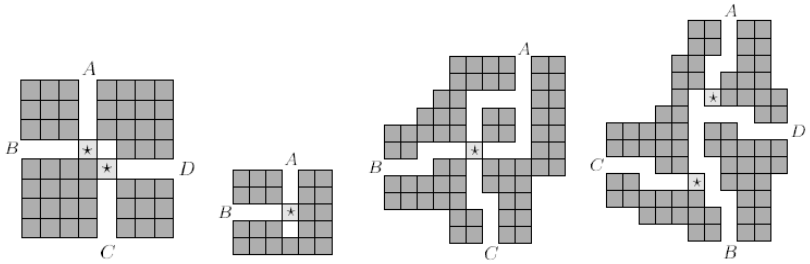
Takto sme zostrojili grafovú konštrukciu, pri prechode ktorej dostaneme 3-ofarbenie grafu. V nasledujúcich riadkoch si ukážeme ako. Na začiatok si predstavme, že ofarbením vrcholu pri jeho opúšťaní ofarbíme aj hranu po ktorej ho opúšťame, a to rovnakou farbou. Zvolíme si teda ľubovolný vrchol. Opustíme ho v smere šípky Eulerovou kružnicou. Pri výstupe z neho narazíme na strojček konzistencie, ktorý nás napojí na inú výstupnú hranu. Sme tak nútený vybrať rovnakú farbu ako už definovaná hrana. V prípade, že hrana bola nedefinovaná postupujeme ďalej až narazíme na farbiacu križovatku, ktorá je prepojená s výstupom vrcholu, do ktorého sa chystáme. Takto sa vylúči možnosť, že zvolíme rovnakú farbu pre dva susedné vrcholy. Teraz máme dosť informácií aby sme korektne ofarbili vrchol a výstupnú hranu. Týmto postupom po celej Eulerovej kružnici postupne definujeme všetky farby vrcholov a hrán a ofarbenie bude korektné. Predveďme si však, ako jednotlivé objekty tohto grafu analogicky previesť na objekty zo sveta hry *SOKOBAN* tak, aby aj naďalej spĺňali požadovanú funkčnosť.

	Jednosmerná skrinka(1,1); je prechodná iba v smere šípky. Akonáhle sa však skrinkou aspoň raz prejde, ostáva naďalej otvorená pre oba smery.
	Vidlicová skrinka(1,3); môže byť opustená ktorýmkoľvek výstupom. Avšak vstúpiť je možné iba do jedného.
	Xor-prechod(2,2); umožňuje prechod dvoma cestami, avšak jednému vstupu je priradený práve jeden výstup, a nemožno vystúpiť iným výstupom
	NAND skrinka(2,2); spája dve cesty tak, že len jedna z nich je priechodná

Obrázok 10: popis funkčnosti jednotlivých použitých skriniek. V zátvorke je uvedený počet vstupov strojčeka a počet výstupov strojčeka.

Skrinka konzistencie a farbiaca križovatka nie sú atomické, ale sa skladajú z menších funkčných dielov. Podobne ako pri dôkaze PSPACE-úplnosti, budeme na ich konštrukciu využívať niektoré postavenia bludiska so špeciálnymi vlastnosťami. Zoznam použitých skriniek možno nájsť na obrázku 10.

Prevodné konštrukcie atomických skriniek zo zoznamu na *SOKOBAN*-skrinky je možné vidieť na obrázkoch.

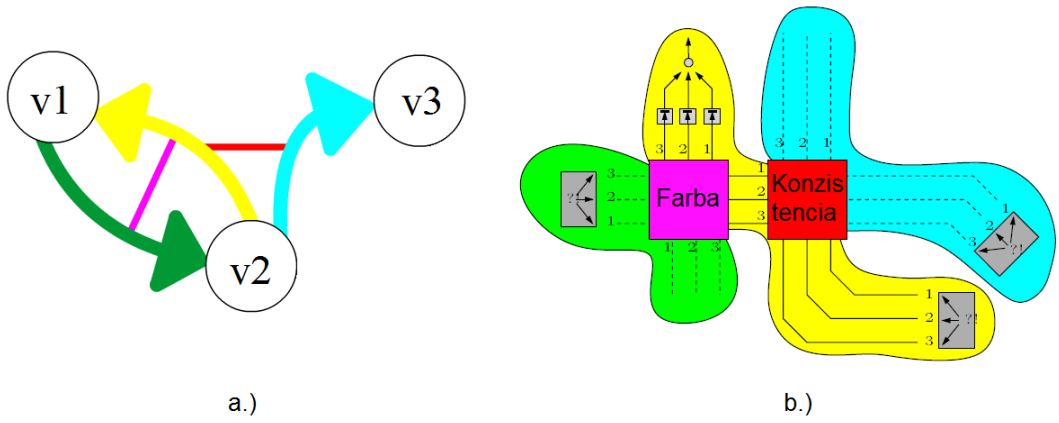


Obrázok 11: Postavenia hry SOKOBAN simulujúce funkčnosť skriniek po poradí zľava doprava: NAND, jednosmerná skrinka, vidlicová skrinka a Xor prechod.

Pre skrinku NAND musí platiť, že pri prechode jednou cestou sa uzavrie druhá. Skutočne, na obrázku pri vstupe do otvoru B a výstupe otvorom A stane sa cesta z C do D neprechodná. Prechodom z C do D sa uzamkne cesta B-A. Funkcia jednosmerného zariadenia je zrejmá. Vstup B je zablokovaný, je možné vojsť iba vstupom A. Prechodom skrinkou sa táto otvorí a zostane otvorená. 3-výstupová vidlicová skrinka v skutočnosti pozostáva z dvoch skriniek uvedených na obrázku 10 ako vidlicová skrinka. Navyše, vstup skrinky A ako aj výstupy skrinky B a C sú v skutočnosti chránené jednosmernou skrinkou. Všimnime si, že prechodom skrinkou typu vidlice a výberom jednej výstupnej cesty nutne uzavrieme druhú. Nie je žiadna možnosť ako

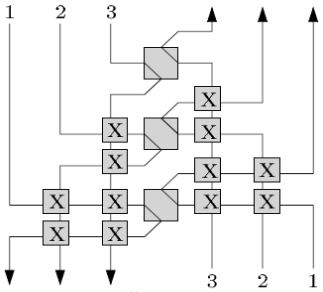
svoje rozhodnutie po tomto kroku zmeniť. Skrinka ostáva prechodná iba jedným výstupom. Poslednou skrinkou je Xor prechod. Vstupné cesty sú označené ako A a C. Vstúpením do jednej z nich a jej opustením cez priradený výstup sa druhá cesta stane neprechodnou. Skutočne, na obrázku vidno, že nie je možné zvoliť inú cestu než priradenú, ako aj fakt, že prejdením uzamkneme druhú cestu.

Vzhľadom na to, že sa pohybujeme vo svete *PUSH-1,* môžeme si predstaviť hranu kružnice *T* ako uličku z prostredia SOKOBAN ohraničenú z oboch strán stenou. V skutočnosti, bude každá hrana reprezentovaná troma takýmito uličkami. Každá z týchto troch uličiek reprezentuje farbu ktorou ofarbíme vrchol z ktorého táto ulička vychádza. Voľba uličky je teda v prenesenom význame voľbou farby opúšťaného vrcholu, a naopak. Voľba farby sa vykoná hneď po výstupe z hrany, no ako vidno z obrázka, pred tým než dôjdeme do ďalšieho vrcholu, môžeme túto hodnotu ešte dvakrát zmeniť, podľa požiadaviek korektného 3-ofarbenia. Tieto uličky budú podľa spomínaného návodu prepojené so skrinkami farby a konzistencie, ako máme možnosť vidieť na obrázku 12. Už sme si ukázali, že takto prepojené hrany nám zaručia korektné ofarbenie grafu.



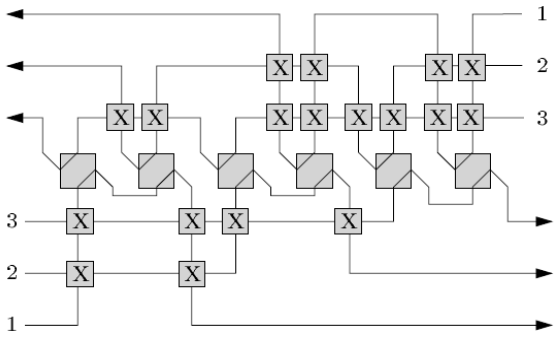
Obrázok 12:Na obrázku a.) je uvedená schéma prepojenia troch vrcholov grafu. Ružová čiara predstavuje križovatku farieb, červená skrinku konzistencie. Na obrázku b.) je tento prípad zapísaný ako SOKOBAN bludisko. Rovnaké farby na obrázkoch a.) a b.) zodpovedajú rovnakým objektom v skutočnosti. Každý koridor (1-3)na obrázku b.) v hrane, reprezentuje jednu farbu jej ofarbenia.

Konštrukcia skrinky konzistencie je implementovaná ako nám to ukazuje schéma na obrázku 13.



Obrázok 13: Skrinka konzistencie. Špeciálne usporiadanie atomických skriniek zaručuje, že pri prvom prechode uzamkneme cesty odlišné od našej a pri druhom prechode strojčekom, si teda musíme vybrať cestu, ktorou sa strojček uzamykal. (volíme rovnakú farbu).

Posledným objektom, ktorého konštrukcia nám ešte nie je známa, je objekt farbiacej križovatky. Na obrázku 14 uvádzam schému ako je možné zostaviť bludisko hry SOKOBAN tak, aby simulovalo činnosť farbiacej križovatky. Tou je zakázanie výberu rovnakého koridoru(farby), ako má protiidúca hrana vedúca cez tento strojček.



Obrázok 14: Skrinka farbiacej križovatky. Vhodný výber a usporiadanie atomických strojčekov zaručuje, že pri prechode prvý krát uzamkneme aktuálnu cestu, čím zamedzíme jej výber pri druhom prechode strojčekom.(vyberáme inú farbu)

Zdá sa, že sme takmer hotoví. Avšak každý problém musí mať začiatok a koniec. Ten náš získame tak, že v ľubovolnom bode narušíme našu kružnicu, čím získame trasu, kde začiatok a koniec definujeme na jej koncoch. Za riešenie problému považujeme, ak hráč dokáže prejsť celú túto cestu od počiatku až do cieľa.

Máme teda k dispozícii algoritmus, ako transformovať problém 3-ofarbenia rovinného grafu na vyriešenie bludiska SOKOBAN. Ukážme si teraz, ako vyzerá samotný dôkaz vety:

Veta *PUSH-1* je NP-ťažký.

Dôkaz Pre zadaný rovinný graf *G(V,E)* použijeme popísaný algoritmus transformácie a vytvoríme *PUSH-1* problém. Veľkosť tohto problému, závisí od počtu strojčekov

v ňom, a ten je zas závislý na počte hrán v *G,* a to lineárne. Teda veľkosť nového problému je najhoršom prípade polynomická vzhľadom k veľkosti grafu *G*. Z čoho plynie polynomická časová zložitosť algoritmu O(p(n)). Neformálne môžeme povedať, že získaný algoritmus transformácie dokáže transformovať problém v polynomickom čase, nakoľko transformuje polynomické množstvo objektov ktorých počet sa transformáciou lineárne zvýši oproti zadaniu. Sčítaním polynómu s polynómom je opäť polynóm, čo je výsledná časová zložitosť.

Ďalším bodom dôkazu, je ukázať vzájomne jednoznačnú riešiteľnosť.

Chystáme sa ukázať, že korektné 3-ofarbenie grafu môžeme dostať iba ak vyriešime SOKOBAN bludisko, ktoré sme získali transformáciou grafu.

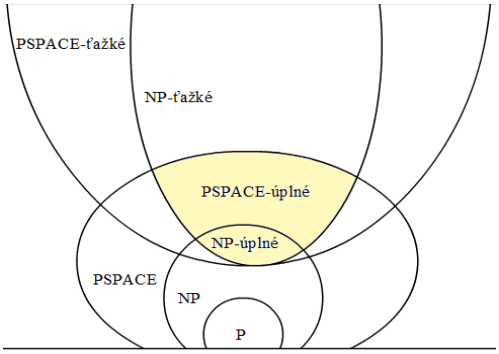
Z konštrukcie skriniek vyplýva, že ak hráč prešiel úspešne do cieľa, tak zakaždým keď opustil vrchol vybral si cestu rovnakej farby akou bol ofarbený pred tým. Ak hráč došiel do cieľa, nemohlo sa cestou stať aby nebola rešpektovaná táto podmienka. Túto vlastnosť „nevyhnuteľnosti“ je možné si overiť priamo zo schémy konštrukcie skrinky konzistencie. Táto „nevyhnuteľnosť“ sa týka taktiež farbiacej križovatky a je možné si ju overiť zo schémy jej stavby. Funkciou $c: V \rightarrow \{1,2,3\}$ nazveme funkciu ktorá je definovaná mapovaním koridorov v bludisku týmito dvoma skrinkami. Na vstupe má vrchol, na výstupe je číslo koridoru vybrané robotom po prechode skrinkami. Funkcia je vďaka spomínaným vlastnostiam skriniek ofarbením. Vyriešením bludiska teda zákonite musíme dostať korektne 3-ofarbený graf.

Na druhú stranu, chceme dokázať, že ak máme 3-ofarbenie $c: V \rightarrow \{1,2,3\}$ zadaného grafu *G,* tak ofarbenie dokáže vyriešiť bludisko zostrojené podľa grafu G popísanou tranformáciou. Funkcia *c(v)* dostane na vstup vrchol a jej výsledkom je číslo 1, 2 alebo 3, čo môžu byť numerické reprezentácie farieb, alebo v našom prípade koridory. Aplikovaním funkcie *c(v)* na vrchol teda dostaneme hodnotu, ktorú budeme považovať za označenie koridora, ktorým sa treba vybrať z vrcholu. Aplikovaním funkcie *c(v)* na všetky vrcholy grafu ofarbíme tento graf, a navyše dopravíme robota bludiskom do cieľového stavu.

Dokázali sme to, čo sme si v úvode kapitoly zadali, a preto prehlasujeme skupinu problémov *PUSH-1* do ktorej spadá aj SOKOBAN za NP-ťažkú.

Dokončením dôkazu sme vymedzili zložitostný priestor nášho problému na prienik NP-

ťažkej množiny úloh s množinou PSPACE-úplných problémov. Vyobrazenie tejto podmnožiny v súvislostiach celého zložitostného priestoru vidíme na obrázku 15.



Obrázok 15: Vyobrazenie zložitostného priestoru hry SOKOBAN. Na obrázku je žltou farbou znázornený prienik množín tak ako sme ho vymedzili dokázanými tvrdeniami

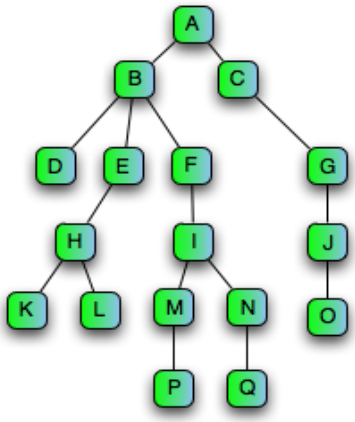
5 Riešenie problémov

Z uvedených dôkazov vyplýva, že nájsť riešenie problému SOKOBAN nebude triviálne. Bežnou metódou riešenia takto obtiažnych úloh je prehľadávanie stavového priestoru, kde stavovým priestorom sa rozumie množina všetkých stavov, do ktorých je možné sa dostať z počiatočného stavu postupnosťou výhradne zostavenou z povolených ťahov[12]. Toto prehľadávanie môže prebiehať viacerými spôsobmi. V tejto kapitole si povieme niečo o dvoch z nich. Prvým z nich je neinformované prehľadávanie, no viac priestoru venujeme zaujímavejšiemu informovanému prehľadávaniu v druhej časti kapitoly.

5.1 Neinformované prehľadávanie

Neinformované prehľadávanie, alebo slepé prehľadávanie, nemá žiadne ďalšie informácie o stavoch, okrem tých, ktoré mu boli poskytnuté pri definícii problému. Jediné čo dokážu tieto algoritmy, je vyrábať následníkov a rozoznávať medzi nimi cieľové a necieľové stavy. Neinformované metódy delíme podľa poradia, v ktorom expandujú stavy, a to na slepé prehľadávanie do hĺbky a slepé prehľadávanie do šírky. Prehľadávanie do šírky ako prvý expanduje vrchol s najmenšou hĺbkou. Hĺbkou *h* uzlu *v* rozumieme počet hrán vedúcich z počiatočného stavu, až k danému uzlu *v*[12].

Prehľadávanie do šírky. Ako prvý expandujeme koreň. Potom expandujeme všetkých jeho následníkov. Potom ich následníkov. Vždy pred tým, než expandujeme vrchol v hĺbke *r*, musíme mať expandované všetky uzly v hĺbke *r-1*. Prehľadávanie do šírky môže byť implementované ako prehľadávanie na strome, kde listy tohoto stromu, jeho okraj, ukladáme do fronty (FIFO). To znamená, že vrcholy, ktoré sme navštívili ako prvé, budeme ako prvé aj expandovať. Beh algoritmu je zobrazený na obrázku 16. Poradie písmen v abecede označuje poradie prehľadávania jednotlivých vrcholov grafu.



Obrázok 16: Neinformované prehľadávanie do šírky. Písmená označujú postupnosť prehľadávania.

Je zrejmé, že daný algoritmus je úplný. Ak je najplytkejší cieľ v hĺbke *d*, tak za podmienky, že vetvenie *b* je konečné, tento cieľ nájdeme po expandovaní všetkých predošlých vrcholov. Samozrejme tento cieľ, ktorý nájdeme, nemusí zákonite byť optimálnym. Slepé prehľadávanie do šírky však môže byť optimálne, a to ak váha cesty je neklesajúcou funkciou hĺbky. Optimálne riešenie teda bude zároveň najplytkejším riešením.

Predstavme si, že máme hypotetický stavový priestor, kde každý stav môže mať až *b* potomkov. Koreň vyprodukuje *b* potomkov, z ktorých každý vyprodukuje ďalších *b* potomkov. Dohromady je to b^2 stavov. Predstavme si teraz, že v našom hypotetickom príklade sa prvý cieľ nachádza v hĺbke *d*. V takomto prípade musíme expandovať všetky vrcholy až k poslednému v hladine *d*. Ľahko si spočítame, že celkovo potrebujeme generovať počet vrcholov rovný

$$b + b^2 + b^3 + ... + b^d + (b^{d+1} - b) = O(b^{d+1})$$

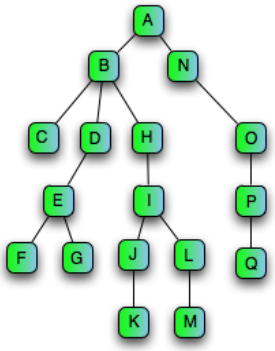
Každý vrchol ktorý prejdeme, musíme uchovať v pamäti, pretože je to buď okrajový vrchol, ktorý bude treba v budúcnosti expandovať, alebo je to nejaký predok takéhoto vrcholu. Z toho vyplýva, že pamäťová náročnosť algoritmu, je rovnaká ako tá časová. Aby sme mali predstavu, o tom či je takto definovaná náročnosť prijateľná, uvádzam tabuľku časov a potrebnej pamäte na vyriešenie niektorých jednoduchých problémov[10].

Hĺbka	Uzly	Čas	Pamäť
2	1100	.11 sekúnd	1megabyte
4	111100	11 sekúnd	106megabytov
6	10^7	19 minút	10gigabytov
8	10^9	31 hodín	1terabyte
10	10^{11}	129 dní	101terabytov
12	10^{13}	35 rokov	10petabytov
14	10^{15}	3523 rokov	1exabyte

Tabuľka 1: Časová a pamäťová zložitosť problému v rôznych hĺbkach.

Prehľadávanie do hĺbky. Pri tomto type prehľadávania sa vždy expanduje koreň s najväčšou hĺbkou. Pri prvom prehľadávaní, sa tak zostupuje stromom až na jeho koniec.

Tento postup možno implementovať pomocou zásobníka(LIFO). Iný postup implementácie je rekurzia, teda funkcia, ktorá volá samu seba z každého potomka.



Obrázok 17: Neinformované prehľadávanie do hĺbky. Písmená označujú postupnosť prehľadávania.

Tento postup má veľmi prívetivé pamäťové nároky. Stačí si totiž pamätať jednu cestu z koreňa do listu, plus okrajových neexpandovaných potomkom vrcholov, ktoré sú obsiahnuté v našej prehľadávanej ceste. Akonáhle preskúmame všetkých potomkov vrcholu, môžeme tento vrchol z pamäte vymazať. Predstavme si opäť hypotetický príklad. Ak použijeme postup prehľadávania do hĺbky, na graf s faktorom vetvenia b a maximálnou hĺbkou m, budeme takto v najhoršom prípade potrebovať uchovať $bm+1$ uzlov. Ak si to prevedieme do praxe, tak v prípade uvedenom v tabuľke 1 v hĺbke $d=12$ nebudeme potrebovať *10 petabytov,* ale postačí nám *118 kilobytov* pamäte[10].

Špeciálnym prípadom prehľadávania do hĺbky je takzvaný *backtracking*. Tento postup má dokonca prívetivejšie pamäťové nároky, než obyčajné prehľadávanie do hĺbky.

Základný rozdiel medzi týmito dvoma postupmi nájdeme, ak sa pozrieme bližšie na to, ako expandujú vrcholy. *Backtrackingu* stačí vždy generovať práve ten vrchol, ktorý sa chystá prehľadávať. Musíme teda pri jeho implementácii myslieť na to, aby si pamätal, ktorého syna má generovať v ďalšom kroku. Pamäťové nároky sa preto týmto spôsobom upravia z pôvodného *O(bm)* na ešte o niečo prívetivejšie *O(m)*. *Backtracking* má ešte jednu výhodu proti obyčajnému hĺbkovému hľadaniu. Generovanie nových potomkov vykonáva tak, že prepisuje popis aktuálneho stavu, miesto toho, aby ho najprv kopíroval a prepisoval až potom. Pre *backtracking* je kritická schopnosť vracať sa o krok spať, aby sme mohli modifikovať nasledujúceho potomka, ak predošlý zlyhá. Pri systémoch, ktoré majú rozsiahly popis stavu, môže byť táto úspora rozhodujúca.

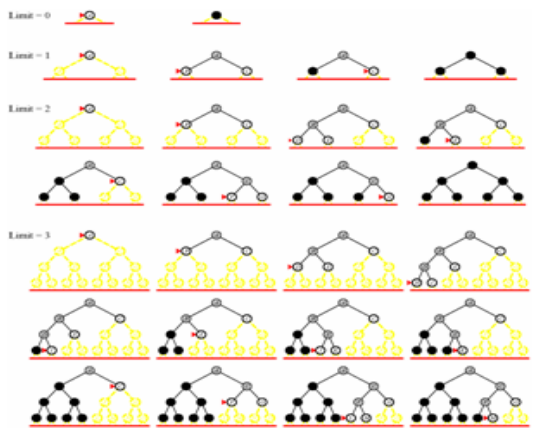
Hlavnou nevýhodou prehľadávania do hĺbky, ktorú zdedil aj *backtracking*, je fakt, že sa jedná tak trošku o tvrdohlavé prehľadávanie. Predstavme si, že riešenie nášho problému leží v neveľkej hĺbke, avšak náš algoritmus si v prvom kroku vyberie cestu, ktorá k nemu nevedie. Stratíme tak veľa času prehľadávaním cesty, ktorá nevedie k cieľu, ba dokonca môže byť nekonečná. Ak aj riešenie nájdeme, nič nám nezaručí, že bude optimálne. Prehľadávanie do hĺbky nezaručuje optimalitu[13]. S úplnosťou je to podobné. Pár riadkov vyššie sme si uviedli príklad, do ktorého sa pri hľadaní riešenia do hĺbky môžeme dostať. Predstavme si teda opäť cestu, ktorá neobsahuje žiaden cieľ, a navyše je nekonečná. Vstúpením na túto cestu sa dostaneme do nekonečného cyklu, ktorý nám nikdy nevráti žiadne riešenie. Preto prehľadávanie do hĺbky nezaručuje ani úplnosť[13].

Medzi neinformovanými prehľadávaniami existuje ešte mnoho variácií na prehľadávanie do šírky a hĺbky, ktoré sa snažia eliminovať niektoré nevhodné vlastnosti týchto algoritmov. My si z nich však povieme niečo viac len o jednom. Základ tohto algoritmu sa totiž využíva aj v algoritmoch z triedy informovaných prehľadávaní, a je preto dobré rozumieť metodike jeho práce.

Tento nový algoritmus je založený na fakte, že nekonečnému prehľadávaniu, alebo prehľadávaniu, ktoré je neprijateľne hlboké, možno obmedziť hĺbku, do ktorej sa môže maximálne ponoriť. Túto maximálnu hĺbku je možné stanoviť buď pevne a nemenne, alebo ako si ukážeme, je možné ju postupom času meniť. Obmedzenie hĺbky so sebou prináša niekoľko problémov. Jedným z nich je, že týmto spôsobom môžeme ešte

rozšíriť neúplnosť nášho algoritmu[12]. Predstavme si, že najbližší cieľ je od počiatku vzdialený d . Čo sa stane ak naša obmedzujúca hodnota bude $l < d$? Samozrejme, že znalosťou problému môžeme vytvoriť pomerne dobré odhady tejto hĺbky. Nejde to však univerzálne, pre akýkoľvek typ problému. Časom prídeme na to, že dobrý odhad hĺbky nenájdeme, kým daný problém nevyriešime[12]. Naskytuje sa nám otázka, prečo sa vôbec takýmto algoritmom zaoberáme. Popísaný algoritmus totiž tvorí teoretický základ pre iný algoritmus, ktorý síce tiež používa obmedzenie hĺbky ponoru, ale nestanovuje túto hĺbku pevne a nemenne.

Algoritmus postupného prehĺbovania. Je to algoritmus prehľadávania do hĺbky s jej obmedzením, ktoré sa postupne zvyšuje. Tento algoritmus kombinuje výhody prehľadávania do hĺbky a do šírky. V knihe Problem Solving Methods od Nillsona[12] je o tomto algoritme vyslovených niekoľko zaujímavých tvrdení, ktoré si ukážeme. Postupné prehĺbovanie má dobré pamäťové nároky $O(bd)$, kde b je vetviaci faktor, d je hĺbka najplytkejšieho vrcholu. Je úplný, za podmienky že vetviaci faktor je konečný, a zaručuje optimalitu riešenia, ak cena cesty je neklesajúcou funkciou hĺbky vrcholu. Na prvý pohľad sa môže zdať, že postupné prehĺbovanie vykonáva mnoho operácií opakovane a zbytočne. Nillson však ukazuje, že toto opakované generovanie rovnakých vrcholov nieje vôbec drahé.



Obrázok 18: Agoritmus postupného prehĺbovania

V prvom rade je treba si uvedomiť, že vo vyhľadávacom strome je väčšina vrcholov v jeho spodnej časti. Predstavme si teda iteratívne prehĺbovacie hľadanie s hĺbkou ponoru d . Vrcholy na hladine d ktorých je najviac, sa generujú raz. Vrcholy

na vyššej hladine sa sa generujú 2 krát, na ďalšej 3 krát, a takto to pokračuje až k potomkom koreňa, ktoré sa generujú d krát. Počet generovaných vrcholov dohromady je teda $N(IP)=(d)b+(d-1)b^2+\ldots+(1)b^d$, čo nám dáva časovú zložitosť $O(b^d)$. V porovnaní so zložitosťou obyčajného prehľadávania do šírky je to stále veľmi prijateľný údaj.

5.2 Informované prehľadávanie

Doteraz sme hovorili o neinformovaných prehľadávaniach. Napriek tomu, že vývojom sme sa dostali ku riešeniu, ktoré by sa na prvý pohľad zdalo byť užitočné, v praxi sa ukazuje, že nasadiť neinformovaný algoritmus náš typ problému, nie je veľmi rozumné. Na druhú stranu, znalosť problému nám umožňuje upravovať jednotlivé algoritmy tak, že dokážu nájsť riešenie ďaleko efektívnejšie, ako ich neinformované varianty.

Informované prehľadávania využívajú znalosti problému, ktoré priamo nevyplývajú z definície problému. Postupom času si ukážeme niekoľko algoritmov, ktoré sú informovanou verziou algoritmov z predchádzajúcej kapitoly.

Základným princípom fungovania informovaných algoritmov je vyberanie si najbližšieho adepta na expandovanie pomocou takzvanej vyhodnocovacej funkcie. Táto funkcia vracia vzdialenosť od cieľa, a je preto výhodné vyberať vždy vrchol s čo najmenšou hodnotou. Ak sa nad týmto hlbšie zamyslíme, uvedomíme si, že ak by sme stále vyberali naozaj najlepší vrchol, miesto hľadania by sme riešili problém ako sa dostať z bodu *a* do bodu *b* po vopred zadanej ceste, čo vlastne nieje žiaden problém. V skutočnosti, je však táto vyhodnocovacia funkcia iba akýmsi odhadom, ktorý nám hovorí, ako veľmi sa nám zdá byť cesta na ktorej sa aktuálny stav nachádza, dlhá, teda ako ďaleko musíme ísť v koreňa do cieľa ak pritom pôjdeme práve týmto stavom. Dokonca sa môže veľmi ľahko stať, že nastane prípad, kedy nás tento odhad zmetie a povedie hľadanie smerom, odkiaľ nemožno očakávať riešenie.

Základným rozlišovacím faktorom informovaných algoritmov je teda vyhodnocovacia funkcia. Aj tú je však možné ďalej rozložiť. Vyhodnocovacia funkcia je sumou vzdialenosti skúmaného vrcholu od počiatku a vzdialenosti vrcholu od cieľa. Vzdialenosť od počiatku býva zväčša určená presne. Naopak kritickým miestom vo vyhodnocovaní odhadovanej cesty, je odhad vzdialenosti aktuálneho stavu od cieľa.

Nateraz nám získané vedomosti o vyhodnocovacej funkcii postačia na to, aby sme boli schopný pochopiť algoritmy ktorých práca je na nej založená. Pozrime sa teda na niekoľko z nich.

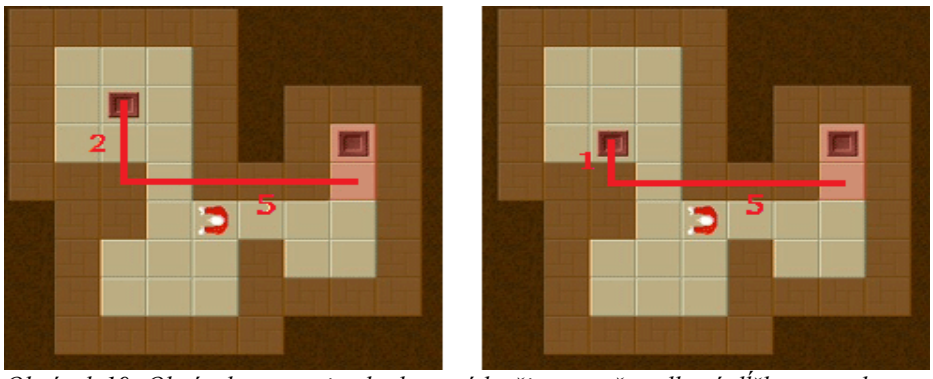
A*. S ohľadom na algoritmus Astar je dobré si zaviesť pojem, o ktorom sme doteraz ešte nehovorili. Jedná sa o pojem prípustnosti algoritmu. Prípustnosť algoritmu v sebe spája vlastnosti úplnosti a optimality. To znamená, že algoritmus je prípustný v prípade, že vždy keď existuje riešenie, tak algoritmus nájde optimálne riešenie[12].

Pre potreby ďalšieho štúdia algoritmu *Astar*, je dobré si pomenovať funkcie vytvárajúce vyhodnocovaciu funkciu. Opäť sa budeme odkazovať na Nillsonovu publikáciu[12], kde okrem definícií možno nájsť tvrdenie týkajúce sa vyhodnocovacích funkcií, ktoré si na konci odstavca vyslovíme. Pre funkciu vzdialenosti od počiatku, sa zavádza značenie $g(n)$, pre funkciu vzdialenosti od cieľa $h(n)$ a pre samotnú vyhodnocovaciu funkciu je to $f(n)$, kde n je aktuálny vrchol prehľadávaného grafu. Prípustnou heuristickou funkciou nazveme takú funkciu $h'(n)$, ktorá nikdy nepresiahne skutočnú hodnotu vzdialenosti vrcholu od cieľa. Vzhľadom na to, že $g(n)$ udáva miesto odhadu presnú hodnotu, prenáša sa tento „optimistický“ údaj vzdialenosti až do funkcie $f'(n)$, o ktorej následne platí, že tiež nepresiahne skutočnú hodnotu cesty vedúcej cez vrchol n . Hodnota funkcie $h'(i)$ musí teda ležať niekde medzi 0 a skutočnou hodnotou $h(i)$. Skutočnú hodnotu $h(i)$ však nepoznáme. Naskytuje sa nám tak otázka, či existuje nejaký spôsob, ako si overiť, či je podmienka prípustnosti pre funkciu $h'(n)$ splnená. Nillson tvrdí, že nie. Jedine znalosťou celého stavového priestoru je možné dokázať prípustnosť heuristiky. Avšak problémy, ktoré to umožňujú, musia byť do značnej miery jednoduché a na riešenie takýchto problémov nie je potrebná žiadna heuristika. Tento fakt sa v praxi ukazuje byť najväčšou slabinou nami práve študovaného algoritmu *Astar*. Útechou nám však môže byť tvrdenie „Graceful Decay of Admisibility“, ktoré zmierňuje požiadavky na prípustnosť heuristiky tým, že povoľuje určitú toleranciu presnosti, a pritom stále dokáže zachovať prípustnosť algoritmu.

Tvrdenie Ak $h'(n)$ niekedy (ale málokedy) nadhodnocuje *h* o hodnotu väčšiu než δ , potom algoritmus *A*(už nejde o *Astar*!) zriedkakedy nachádza riešenie, ktorého cena je väčšia o viac než δ v porovnaní s cenou skutočne optimálnej cesty.

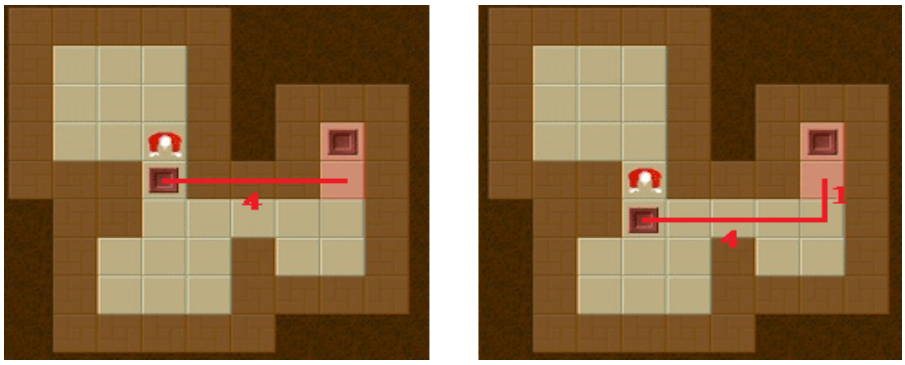
Jednoduchým príkladom heuristiky, je heuristika Manhattan. Tá sa spolieha na to, že

problém je situovaný do plochy tvaru štvorcovej siete. Na obrázku 19 vidieť, ako Manhattan funguje.



Obrázok 19: Obrázok vpravo je ohodnotený lepšie, pretože celková dĺžka cesty do cieľa je kratšia než na obrázku vľavo

Situácia na pravom obrázku je ohodnotená ako perspektívnejšia, vzhľadom na to, že celková vzdialenosť od cieľa je menšia. Je zrejmé že Manhattan nikdy odhadne cestu na kratšiu vzdialenosť ako je to v skutočnosti. Manhattan je preto bezpečná, avšak nie veľmi účinná heuristika. Na ďalšom obrázku si ukážeme prečo. Na ľavom obrázku je ťah ohodnotený lepšie ako ťah na pravom obrázku, pričom skutočnosť je iná.



Obrázok 20: Situácia na ľavom obrázku je napriek skutočnosti ohodnotená lepšie.

Ako sme naznačili v úvode kapitoly o algoritme Astar sa vie, že je prípustný. Aby sme sa na to však mohli spoľahnúť pri riešení problémov, musíme mať toto tvrdenie dokázané. Preto sa teraz pozrieme, ako sa tento dôkaz konštruuje[12]. Tvrdíme teda, že ak je heuristická funkcia prípustná, potom je prípustný aj algoritmus, ktorý s ňou pracuje. Pri dokazovaní budeme postupovať tak, že najprv ukážeme, že predtým než algoritmus *Astar* ukončí svoju činnosť, bude vždy existovať nejaký uzol medzi uzlami

pripravenými na preskúmanie a na optimálnej ceste, ktorej f' hodnota nie je väčšia ako skutočná hodnota optimálnej cesty $f'(s)$. Z uvedeného vyplýva, že prípustnosť algoritmu vyplýva z faktu, že uzol v zozname, kde sa nachádzajú uzly pripravené na preskúmanie, ktorý má minimálnu f' hodnotu, nemôže byť cieľovým vrcholom, a to až do chvíle, kým sa nenájde cieľový vrchol, ktorý má f' hodnotu rovnú $f'(s)$. Najprv teda musíme dokázať lemma:

Lemma 4.2 Ak $h'(n) \leq h(n)$ pre všetky n , tak kedykoľvek predtým než skončí *Astar* a pre akúkoľvek optimálnu cestu *P* z uzlu *s* do cieľa, existuje otvorený uzol n' na trase *P* s $f'(n') \leq f(s)$

Dôkaz Predpokladajme, že optimálna cesta je reprezentovaná usporiadanou postupnosťou $(s = n_0, n_1, n_2, \ldots, n_k)$, kde n_k je cieľový vrchol. Nech n' je prvý otvorený vrchol v postupnosti. Určite existuje aspoň jeden taký vrchol, pretože keby n_k bolo medzi uzavretými, *Astar* by skončil. Podľa definície f' máme

$f'(n') = g'(n') + h'(n')$.

Vieme, že *Astar* našiel optimálnu cestu do n' , keďže n' je na *P* a všetci jeho predchodci na *P* sú už uzavretí. Preto platí

$f'(n') = g(n') + h'(n')$.

Vzhľadom na to, že predpokladáme $h'(n') \leq h(n')$, môžeme napísať $f'(n') \leq g(n') + h(n') = f(n')$.

Ale f hodnota každého uzlu na optimálnej ceste je rovná $f(s)$, minimálnej cene, a preto dostávame $f'(n') \leq f(s)$, ako sme chceli dokázať.

Teraz musíme ešte dokázať, že ak $h'(n)$ je dolnou hranicou $h(n)$ tak *Astar* je prípustný. Teda

Teorém 4.3 Ak $h'(n) \leq h(n)$ pre všetky vrcholy n a ak všetky ceny krokov sú väčšie ako nejaké malé kladné číslo δ , tak algoritmus Astar je prípustný.

Dôkaz Použijeme dôkaz sporom. Tento spor bude vyplývať z toho, že budeme predpokladať, že *Astar* neskončí vždy nájdením optimálnej cesty z počiatku do cieľa. Musíme sa teda zamyslieť nad troma možnosťami ktoré môžu nastať. Jedna je, že

algoritmus sa zastaví bez toho, aby našiel nejaký cieľový stav. Druhá, nezastaví sa vôbec. Tretia, zastaví sa v cieľovom uzle, ktorého dosiahnutie však nebude najlacnejšie.

Prípad 1: Ukončenie hľadania bez nálezu cieľového stavu. Ak sa pozorne pozrieme ako algoritmus *Astar* funguje, zistíme, že situácia v ktorej sa má algoritmus zastaviť, je podmienená nájdením cieľového stavu. Jediný prípad kedy sa tak môže stať je ten, keď sa zoznam uzlov na preskúmanie vyprázdni, a my už nebudeme mať ďalej čo expandovať. Z predchádzajúcej lemmy 4.2 vieme, že tento zoznam sa nemôže vyprázdniť pred ukončením algoritmu ak existuje aspoň jedna cesta z počiatku do cieľa. Inými slovami, do tejto situácie sa nedostane, jedine že zadaný problém nemá riešenie.

Prípad 2: Algoritmus sa vôbec nezastaví. Nech t je cieľový stav, dosažiteľný z počiatku konečným počtom krokov s priradenou minimálnou cenou $f(s)$. Vzhľadom na to, že cena každého kroku je najmenej δ , tak pre každý uzol ďalej než $M = f(s)/\delta$ krokov z s máme $f'(n) \geq g'(n) \geq g(n) > M\,\delta = f(s)$. Je zrejmé, že žiaden uzol ďalej než M krokov od počiatku nebude preskúmaný. Podľa lemmy 4.2 existuje nejaký uzol n' v zozname OPEN, ktorý leží na optimálnej ceste, taký že $f'(n') \leq f(s) < f'(n)$, takže podľa algoritmu Astar vyberie n' miesto n . Jediným dôvodom, prečo *Astar* neukončí svoju činnosť takto, môže byť postupné znovuotváranie uzlov, ktoré sa nachádzajú do vzdialenosti M od počiatku s . Buď $\chi(M)$ množina uzlov dosiahnuteľných z počiatku M krokmi, a buď $v(M)$ počet uzlov v $\chi(M)$. Každý uzol n v $\chi(M)$ môže byť otvorený najviac konečný počet krát, povedzme $\rho'(n, M)$ kvôli tomu, že existuje iba konečný počet ciest z s do n , ktoré prechádzajú iba cez uzly vzdialené M a menej krokov od s . Nech $\rho(M) \max_{n \in (M)} \rho'(n, M)$ je maximálny počet otvorení ktoréhokoľvek uzlu. Z toho vyplýva, že maximálne po $v(M)\rho(M)$ expandovaniach všetky vrcholy v $\chi(M)$ musia byť nadobro uzavreté. Nakoľko žiaden uzol mimo množiny $\chi(M)$ nemôže byť expandovaný, *Astar* musí skončiť.

Prípad 3: Ukončenie činnosti algoritmu v cieľovom stave bez nájdenia minimálnej cesty. Predpoklad teda je, že *Astar* skončí v cieľovom stave t s $f'(t) = g'(t) > f(s)$. Ale podľa uvedenej lemmy 4.2 existoval tesne pred ukončením algoritmu otvorený uzol n' , ktorý ležal na optimálnej ceste s $f'(n') \leq f(s) < f'(t)$, takže vlastne by bol

miesto t vybraný na expandovanie uzol n' , ktorý by nám dal spor s podmienkou, že *Astar* ukončil hľadanie.

Dôkaz teorému môžeme týmto prehlásiť za hotový.

V ďalších odstavcoch si ukážeme, že heuristická funkcia má, obmedzená ešte jedným požiadavkom, veľmi zaujímavú vlastnosť, ktorú si tu však nebudeme krok po kroku dokazovať, avšak aspoň naznačíme, akým by sa taký dôkaz mal uberať smerom.

Najprv si však povieme niečo o vlastnostiach a účinkoch výberu zložiek $f(n)$ na výkonnosť algoritmu. Akú má teda úlohu $g(n)$? Táto funkcia okrem nádejnosti z hľadiska toho ako rýchlo je možné dosiahnuť cieľ, umožňuje brať do úvahy aj hodnotenie, ako výhodná bola cesta z počiatku do cieľa. Z hľadiska optimalizácie je toto veľmi dôležitý krok. Ak je pre nás prvoradé nájdenie nejakého riešenia, akéhokoľvek riešenia problému, položíme $g=0$.

S ohľadom na možnosti výberu funkcie $h'(n)$ sa budeme musieť ponoriť do hlbších úvah. Zopakujme si, čo sa stane v prípade, že položíme $h'(n)=h(n)$. Algoritmus bude expandovať priamo uzly na optimálnej ceste a k žiadnemu skutočnému hľadaniu nikdy nedôjde. Ďalšia zaujímavá možnosť, ktorá stojí za zmienku je tá, keď položíme $h'(n)=0$ pre všetky n . Takto fungujúce hľadanie sa nazýva algoritmus s rovnomernou cenou (uniform-cost search)[10]. Hodnotu funkcie $f(n)$ určuje iba z hodnoty $g(n)$. Aj takto ladený algoritmus však môže zaručovať za istých okolností optimalitu. My sa však presunieme k inému algoritmu.

Pozrime sa ako pracuje algoritmus vetiev a medzí (branch-and-bounds)[10]. Do istej miery je možné ho chápať ako rozšírenie algoritmu usporiadaného prehľadávania. Funguje totiž rovnako až do chvíle, než nájde nejaký cieľový stav. V tú chvíľu si zapamätá cenu cesty z počiatku do tohto cieľového stavu, no svoju prácu tu nekončí. Pokračuje ďalej, no zo zoznamu OPEN automaticky vyhadzuje uzly, ktoré majú vyššiu cenu než zapamätaná cena. Ak nájde riešenie s nižšou cenou zapamätá si novú cenu a pokračuje ďalej v hľadaní. Algoritmus sa zastaví, až sa vyprázdni zásobník OPEN. Tento algoritmus teda doprehľadáva celý stavový priestor, čo mu síce zaručuje, že nakoniec nájde optimálne riešenie, avšak prehľadáva aj zbytočné uzly, ktoré nevedú do cieľa. Stačí, že majú nižšiu cenu než aktuálne riešenie. Jedná sa však

o neinformovaný algoritmus, ktorý nemá žiadne znalosti problému než tie, ktoré sú uvedené v definícii, a ktoré si zistil sám. Ak položíme okrem $h'(n)=0$ ešte aj $g(n)=0$ to, čo nakoniec dostaneme, bude algoritmus náhodného prehľadávania, teda verzia neinformovaného prehľadávania. Ak ohodnotíme krok cenou 1, tak že $g(n)$ bude vyjadrovať hĺbku uzlu, vrátime sa k algoritmu prehľadávania do šírky.

Na rôznych heuristickým funkciách, je možné nájsť istý typ usporiadania. Nás bude zaujímať to ktoré hovorí, že algoritmus A je informovaný viac, než algoritmus B, ak heuristika použitá v A umožňuje spočítať dolnú hranicu $h(n)$ tak, že táto hranica je v každom prípade, keď sa nejedná o cieľový vrchol, ostro väčšia než dolná hranica dopočítaná heuristikou použitou v B[10]. Samozrejme, nie v prípade každého problému je efektívne sa zaoberať počítaním čo najpresnejšej heuristiky vzhľadom na to, že takto presný odhad ide často ruku v ruke s vyššími výpočtovými nárokmi na systém. Je preto dobré zvážiť heuristiku, ktorá berie do úvahy oba faktory a dokáže sa rozhodovať pomerne presne a v rozumnom čase. Teraz už máme dosť informácií, aby sme sa mohli vrátiť opäť k algoritmu *Astar* a ukázať si jeho vlastnosť, o ktorej som hovoril pár riadkov vyššie.

Táto vlastnosť zaručuje, že ak funkciu $h'(n)$ mierne zaťažíme ďalším pravidlom, stane sa priamo *Astar* optimálnym a nielen optimalitu zaručujúcim (je treba dôsledne odlišovať tieto vlastnosti) algoritmom. Inými slovami, Algoritmus Astar nikdy neexpanduje viac uzlov, než akýkoľvek iný prípustný algoritmus A taký, že Astar je informovaný viac než A. Tvrdenie aj výrok boli vyslovené v publikácii[12], z ktorej budeme čerpať návod konštrukcie dôkazu.

Aby sme toto tvrdenie mohli dokázať potrebujeme, uvážiť nejaký algoritmus A taký, že algoritmus Astar by bol viac informovaný ako A. Ukázali by sme, že každý vrchol, ktorý expanduje Astar algoritmus, musí byť expandovaný aj algoritmom A. Aby sme to však mohli spraviť, potrebovali by sme ukázať, že akýkoľvek vrchol, ktorý Astar expanduje, podlieha dvom nariadeniam. Jedným z nich je, že pre akýkoľvek vrchol ktorý Astar expanduje v tom čase už je známa optimálna cesta z počiatku do tohto vrcholu $g'(n)=g(n)$. Druhým je, že ak Astar expanduje nejaký vrchol, tak hodnota $f'(n)$ nie je väčšia ako hodnota $f(s)$. Pomocou týchto nariadení môžeme ukázať, že ak A neexpanduje nejaký vrchol ktorý Astar expandoval, tak A musel vedieť, že

žiadna cesta do cieľa ktorá obsahuje tento vrchol, nie je optimálna. Inými slovami, Astar nemôže byť viac informovaný ako algoritmus A v prípade, že A si môže dovoliť neexpandovať vrcholy, ktoré Astar expanduje. V prvom rade je teda treba sa presvedčiť o funkčnosti nariadení spomínaných vyššie. V prípade, že hľadanie prebieha na stromovej štruktúre, je prvá časť pomerne triviálna vzhľadom na to, že v strome existuje pre každý uzol jediná cesta začínajúca v počiatku a vedúca práve do tohto vrcholu. Uvažujúc ale všeobecný prípad, budeme potrebovať položiť nejaké obmedzenia aj na $h'(n)$, aby sme zaručili, že zakaždým keď budeme expandovať nejaký vrchol v grafe budeme v tej chvíli už poznať optimálnu cestu do tohto vrcholu. Predpokladajme, že pre ľubovolné dva vrcholy m a n pre ktoré existuje $k(m,n)$

$h'(m)-h'(n)\leq k(m,n)$ čo znamená, že rozdiel medzi dvoma odhadnutými hodnotami vzdialeností dvoch ľubovolných vrcholov od cieľa musí byť menší než optimálna cesta vedúca z jedného vrcholu do druhého. Táto nová požiadavka typicky nebude porušená, ak heuristická informácia, podľa ktorej počítame $h'(n)$ platí rovnomerne pre všetky vrcholy grafu. Tento predpoklad sa nazýva predpoklad rovnomernosti.

Pomocou predpokladu rovnomernosti môžeme dokázať, že ak *Astar* expanduje vrchol, tak už pozná optimálnu cestu do tohto vrcholu. Tento fakt sa ďalej využije na dokázanie optimality algoritmu *Astar*. Za druhé zaručuje, že *Astar* nikdy nebude znovu potrebovať vrchol zo zoznamu uzavretých vrcholov vzhľadom na to, že optimálnu cestu do tohto vrcholu už pozná.

Teraz si uvedieme lemmy a teorémy, ktoré sú potrebné k dôkazu, ale ktoré už jednotlivo dokazovať nebudeme.

Lemma 4.4 Nech je naplnený predpoklad rovnomernosti. A nech stav n je uzavretý algoritmom *Astar*. Potom $g'(n)=g(n)$.

Lemma 4.5 Pre akýkoľvek stav n uzavretý algoritmom *Astar*, ak $h'(n)$ je dolným odhadom $h(n)$ tak $f'(n)\leq f(s)$.

Teorém 4.6 Nech *A* a *Astar* sú prípustné algoritmy také, že *Astar* je viac informovaný ako *A*, a nech je splnený predpoklad konzistencie, funkciou $h'(n)$ použitou v *Astar*. Potom pre každý graf, ak stav n bol expandovaný algoritmom *Astar*, bol tiež

expandovaný algoritmom *A*.

Týmto sme dokázali, že *Astar* je optimálny algoritmus. Nevýhodou algoritmu *Astar* je pomerne problémová pamäťová náročnosť. Nehodí sa teda na riešenie rozsiahlych problémov. Existujú však rôzne variácie tohto algoritmu, kde bola táto náročnosť znížená na prijateľnejšiu úroveň. Napriek tomu tieto algoritmy netrpia stratou optimality alebo úplnosti.

Jedným z týchto algoritmov, je algoritmus *IDAstar[10]*. V časti o neinformovaných algoritmoch sme sa zaoberali jedným algoritmom, o ktorom sme tvrdili, že sa používa v informovaných prehľadávaniach. Práve *IDAstar* je algoritmom používajúcim myšlienku iteratívneho prehĺbovania. Celý jeho názov je *Iterative Deeping Astar*. Tak ako som už teda spomínal, hlavná výhoda iteratívnych prehĺbovacích algoritmov spočíva v úspore pamäti. Zmena informovanej verzie proti tej neinformovanej je v spôsobe, ako sa jednotlivé cesty orezávajú. Miesto neinformovaného orezávania ciest podľa hĺbky vrcholov v strome používa informovaná verzia orezávanie na základe $f(n)$ hodnoty (odhad vzdialenosti cieľa). To znamená, že pri prezeraní vrcholu sa zistí táto $f(n)$ hodnota, ktorá sa porovná s aktuálnou hodnotou medze, a ak je táto hodnota väčšia, algoritmus zastaví prehľadávanie tejto vetvy. Každou iteráciou sa medza zvyšuje o minimálnu hodnotu, o ktorú bola v predchádzajúcom kroku prekročená. Ak algoritmus využíva prípustnú heuristickú funkciu, zaručuje nález optimálneho riešenia.

6 Praktická časť

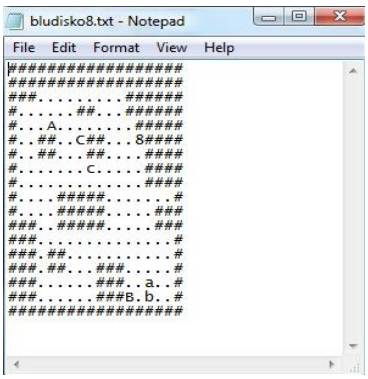
Súčasťou tejto práce by mali byť aj praktické výsledky činnosti niektorých rozoberaných algoritmov. Preto spolu s ňou vznikol program SokobanSolver. Tento program implementuje tri z nich. Sú to Astar, IDAstar, a aby sme mohli otestovať význam používania informovaných algoritmov, tak som implementoval aj predstaviteľa neinformovanej skupiny, backtracking. Pre porovnanie som v programe použil nielen jednu, ale dve heuristiky, obe pre každý informovaný algoritmus. Týmito heuristikami sú manhattan a dijkstrovo hľadanie najkratšej cesty v grafe. O jednotlivých heuristikách si ešte povieme bližšie na konci kapitoly. Taktiež som vyskúšal dva druhy kontajnerov na udržiavanie vrcholov pripravených na preskúmanie, takzvaného zoznamu *OPEN*.

Aby sme mohli riadne otestovať ako algoritmy tak aj samotný program SokobanSolver, vznikol ešte podporný program SokobanGenerator. Tento program má na starosti náhodné generovanie SOKOBAN bludísk. Používa pritom systém prezentovaný v práci Automatic Making of SOKOBAN Problems[14]. Tento systém nám umožňuje dosahovať lepšie výsledky pri generovaní algoritmov v podobe väčšieho počtu riešiteľných bludísk. V ďalšej časti sa teda pozrieme ako tento program funguje. Potom sa vrátime k SokobanSolveru a nakoniec si ukážeme výsledky testu vykonaného na týchto dvoch programoch.

6.1 SokobanGenerator

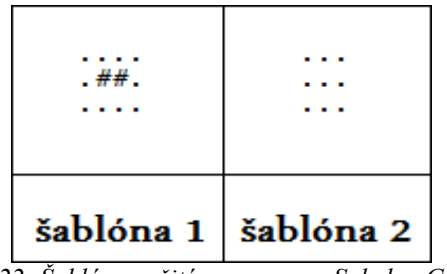
Ako sme už spomínali, SokobanGenerator je podporným programom aplikácie SokobanSolver. Jeho úlohou je vyprodukovať veľké množstvo náhodných bludísk a čo najviac sa pokúsiť zvýšiť šancu riešiteľnosti bludiska.

Výstupom programu SokobanGenerator je adresár so zadaným počtom vygenerovaných textových súborov. Program som testoval maximálne pre hodnotu 10000 vygenerovaní. Názvy súborov sú generované tiež automaticky tak, aby ich mohol následne spracovať SokobanSolver. Obsah jedného súboru nesie obraz hracieho plánu tak, ako ho môžeme vidieť na obrázku 21.



Obrázok 21: Výstup programu SokobanGenerator.

Ako som už spomínal, SokobanGenerator negeneruje náhodne objekt po objekte, ale používa pokročilejší systém šablón. Tento systém je popísaný v práci zverejnenej kolektívom pána Muraseho[14]. Momentálne je program napísaný tak, že dokáže pracovať s 2 šablónami. Šablóny použité pri našom teste sú na obrázku 22. Princíp tohto postupu je nasledovný. Vytvorí sa objekt bludisko, ktorý obsahuje jednak dvojrozmerný kontajner(hrací plán) a za druhé, obsahuje informácie o počte krabíc, stien, cieľov a informácie o súradniciach hráča. Na začiatku je celý hrací plán pokrytý stenami. Postupom času sa nechá plán pokrývať šablónami. Prvá šablóna je položená náhodne. Ďalšia už rešpektuje podmienku vzájomného prieniku s prvou šablónou. Z tohto dôvodu je v programe implementovaná metóda *haveInCommon()*, vracajúca *true* v prípade prieniku. Takto sa zamedzí vzniku odrezaných miestností v bludisku. Výber šablóny je riadený prvkom náhody. Po každom preložení sa overuje hodnota počtu stien s tou zadanou. Ak hodnota v bludisku prvý krát podlezie zadanú hodnotu, generovanie bludiska sa ukončí. Je však zrejmé, že vo vygenerovanom bludisku bude počet stien súhlasiť so zadaným požiadavkom iba vďaka tolerancii malej odchýlky.



Obrázok 22: Šablóny požité v programe SokobanGenerator

Po vygenerovaní bludiska prichádza na rad umiestnenie krabíc. Toto sa však deje

prostredníctvom usadenia cieľových stavov a následného hľadania možných počiatkov pre jednotlivé ciele. Usádzanie cieľov prebieha náhodne. Avšak pri každom pokuse sú kontrolované základné požiadavky na riešiteľnosť. Cieľ sa nesmie prekrývať so stenou, musí byť obsiahnutý v hracom pláne a okolo neho musí byť voľné miesto. Až sú usadené všetky ciele, môžeme začať pokladať krabice. Postupne pre každý cieľový stav určíme všetky miesta odkiaľ je možné sa dostať do tohto stavu. Túto informáciu si vediem pomocou kópie hracieho plánu, možné miesta označujem hviezdičkou. Aj tu kontrolujem prekrývanie sa so stenou. Nakoniec miesto pre krabicu vyberieme opäť náhodným spôsobom, pričom hádame číslo v medziach hracieho plánu. Následne overujeme, či sa na danom mieste nachádza hviezdička. Toto hádanie sa ukázalo byť v niektorých okrajových prípadoch chybové. Dochádzalo k tomu, že pre niektoré cieľové stavy je priestor natoľko malý, že dochádzalo k dosť veľkému plytvaniu časom, miestami pretekaniu pamäte. Obmedzil som preto toto hádanie na 400 pokusov, pričom po ich prekročení sa bludisko zahodí a vytvorí sa nové.

Nakoniec sa určí poloha hráča. Je taktiež určená náhodne. Overujem či táto poloha nekoliduje s ostatnými objektami v hre.

Uvedený postup vytvorí jedno bludisko ktoré sa vypíše do textového súboru a uloží do adresára. Program potom zmení nastavenia vstupných hodnôt. Mení sa veľkosť bludiska a to v rozmedzí dĺžky steny od 10 do 19, mení sa aj počet krabíc v rozmedzí od 1 do 3 a poslednou meniacou hodnotou je hustota zastavania v rozmedzí 20 do 50 percent s veľkosťou kroku 10 percent. Tieto hodnoty sú nastavené pre účely testu.

Hodnota veľkosti plochy nie je v programe obmedzená, obmedzuje ju však pamäťová charakteristika systému na ktorom program beží. Počet krabíc je, s ohľadom na to, že krabice sú označované písmenami abecedy, ohraničený počtom znakov abecedy. Obsah zastavanej plochy samozrejme môže byť v rozmedzí 0 až 100 percent, pričom pri zadaní druhého uvedeného údaju sa položí práve jedna šablóna.

6.2 SokobanSolver

Na úvod kapitoly sa pozrieme, ako sa program SokobanSolver používa. Následne sa pozrieme aj na jeho vnútornú štruktúru z programátorského hľadiska.

6.2.1 Užívateľský manuál

SokobanSolver používa pre získavanie informácií od užívateľa argumenty príkazovej riadky. Prvá sa teda volá funkcia, ktorá celý vstup rozdelí do dátových štruktúr používaných programom.

Prvým argumentom príkazového riadku je názov programu. Tento argument, nebudeme potrebovať, preto ho preskočíme.

Program dokáže pracovať s viacerými algoritmami ktoré kombinuje s viacerými heuristikami, preto si musíme zvoliť dvojice algoritmus-heuristika, ktoré chceme na bludiská aplikovať. Značka začiatku zadávania algoritmov je „-A“. V prípade že na príkazovom riadku bude táto značka chýbať hneď za názvom programu, znamená to, že vstup nie je správne naformátovaný a program musí skončiť. Na konzolovom výstupe sa objaví správa o chybe a možnostiach jej odstránenia. Možnosťami odstránenia sa myslí, že program navrhne možnosti, ktoré na vstupe očakáva. Nasleduje údaj o prvom použitom algoritme. Program očakáva na vstupe jednu z možností: „AstarSet“, „AstarHeap“, „IdaStar“, „Backtracking“. Pri akejkoľvek zámene vstupných údajov so zmienenými, program ohlási chybu, navrhne riešenie pre jej odstránenie a skončí. Význam jednotlivých argumentov je nasledujúci:

- „AstarSet“ bludisko sa bude riešiť algoritmom využívajúcim dátovú štruktúru „set“ definovanej v knižnici <set>.
- “AstarHeap” bludisko sa bude riešiť algoritmom Astar využívajúcim dátovú štruktúru haldy.
- „IdaStar“ bludisko sa bude riešiť algoritmom IDAstar.
- „Backtracking“ bludisko sa bude riešiť algoritmom backtracking.

Po tomto údaji nasleduje ďalší povinný údaj, ktorým je použitá heuristika. Program na vstupe očakáva jednu z možností: „manhattan“, „dijkstra“, „zero“, „none“. Pri akejkoľvek zmene vstupných údajov, proti uvedenému vzoru program ohlási chybu, navrhne očakávaný vstup a skončí. Význam uvedených argumentov je nasledujúci:

- „manhattan“ - použije sa heuristika manhattan
- „dijkstra“ - s algoritmom sa použije heuristika dijkstra

- „zero“ - nepoužije sa heuristika, miesto heuristického odhadu sa dosadí 0
- „none“ - tento argument sa používa spolu s neinformovaným algoritmom backtracking, požitie tohto argumentu spolu s informovaným prehľadávaním je ilegálne a povedie k tomu, že program ohlási chybu na konzolový výstup, a ukončí svoju činnosť.

Program SokobanSolver umožňuje použiť pri jednom spustení viacero dvojíc algoritmus-heuristika, ktoré budú na prehľadávané bludisko aplikované v sérii za sebou. Nasledujúci argument teda môže mať dva tvary. Jedným tvarom ktorým je algoritmus sa dostaneme na počiatok tejto kapitoly. Program prijíma dvojice algoritmus-heuristika do chvíle kým vyhovujú zadaným pravidlám, alebo kým užívateľ nedá najavo, že so zadávaním algoritmov skončil. Počet zadaných algoritmov nieje programom zhora obmedzený. Tento vstup je však obmedzený zdola. V prípade, že program zistí absenciu aspoň jednej dvojice algoritmus-heuristika, pošle na konzolový výstup chybové hlásenie, a skončí.

	manhattan	dijkstra	zero	none
AstarSet	ANO	ANO	ANO	NIE
AstarHeap	ANO	ANO	ANO	NIE
IdaStar	ANO	ANO	ANO	NIE
Backtracking	NIE	NIE	NIE	ANO

Obrázok 23: Možné dvojice algoritmus-heuristika. "Ano" označuje možnú dvojicu, "Nie" označuje nepovolenú dvojicu

Druhým tvarom nasledujúceho argumentu je „-F“. Prítomnosť tohto argumentu na príkazovom riadku je povinná. Užívateľ takto signalizuje, že ukončil zadávanie algoritmov a heuristík a začína zadávať zdrojové súbory. V prípade, že užívateľ ukončí zadávanie argumentov na príkazovom riadku bez príznaku „-F“, program pošle na výstupnú konzolu chybové hlásenie, navrhne možné riešenia a ukončí činnosť. Za týmto príznakom program očakáva výraz v úvodzovkách ktorý označuje názov súboru. Funkcia musí rozoznať aj názov súboru alebo adresára pomocou zadaného výrazu obsahujúce metaznaky „*“ a „?“. Význam týchto metaznakov je nasledujúci:

- * : znak hviezda(asterisk) nahrádza ľubovolný počet výskytov ľubovolného znaku v názve. Ľubovolný počet znamená aj nulový počet, teda absenciu akéhokoľvek znaku.

- ? : znak otáznik(question mark) nahrádza jeden alebo žiaden ľubovolný znak v názve.

	*.?	b?????.txt
bludisko	NIE	NIE
bludisko.	ÁNO	NIE
bludisko.txt	NIE	NIE
blud.txt	NIE	ÁNO
bludisko.t	ÁNO	NIE
b.txt	NIE	ÁNO
bludis.txt	NIE	ÁNO
.txt	NIE	NIE

Obrázok 24: Tabuľka demonštruje príklad použitia metaznakov, a ich význam. V prvom riadku sú uvedené výrazy s metaznakmi. V prvom stĺpci sú porovnávané súbory. Výplňou tabuľky sú výsledky porovnania názvu s výrazom. "Áno" znamená že výraz sa s názvom zhoduje. "Nie" znamená opak.

V prípade, že funkcia nájde podľa zadaného výrazu položku o ktorej zistí, že je to súbor, pridá tento súbor do zoznamu spracovaných súborov. V prípade, že funkcia nájde podľa zadaného výrazu položku o ktorej zistí, že je to adresár, načíta všetky súbory v adresári obsiahnuté. Rekurzívne, ak v otvorenom adresári zistí ďalší adresár, tento tiež otvorí a zaradí do zoznamu všetky obsiahnuté súbory. Obsah súborov nekontroluje. Program prijíma zhora neobmedzené množstvo vstupných výrazov, no podobne ako pri zadávaní algoritmov, je nutné zadať aspoň jeden vstupný súbor. V opačnom prípade program, pošle na konzolový výstup chybové hlásenie a ukončí svoju činnosť.

Zadávanie vstupných súborov ukončí príznak „-K“, ktorý je nepovinný a určuje výstupný súbor, do ktorého budú zapisované údaje o počte preskúmaných stavov na ceste do koncového stavu. Názov súboru je nutné uvádzať v úvodzovkách. Druhou možnosťou ako ukončiť zadávanie vstupných súborov je príznak „-C“, ktorý je taktiež nepovinný a určuje výstupný súbor, do ktorého bude program ukladať informácie o dobách potrebných k tomu, aby sme pre konkrétne bludisko našli riešenie. Napriek

tomu, že oba posledne menované argumenty sú každý samostatne nepovinné, na vstupe programu musí byť definovaný aspoň jeden z nich. V prípade, že užívateľ chce definovať oba údaje, musí tak učiniť v poradí „-K“ + názov súboru a až následne „-C“ + názov súboru.

Vstup vytvorený podľa vyššie uvedených pravidiel je korektný a môže slúžiť ako vstup programu SokobanSolver. Akékoľvek ďalšie argumenty nad rámec tohto korektného vstupu budu ignorované.

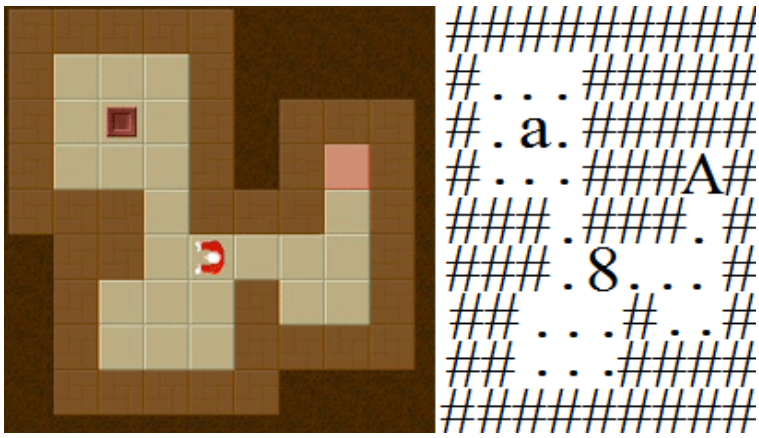
Formát bludiska

Každé načítané bludisko musí spĺňať predkladaný formát. Reprezentácia jednotlivých prvkov bludiska je na obrázku 25.

#	Mriežka označuje stenu
.	Bodka označuje voľnú plochu
8	Číslica 8 označuje polohu hráča
x	Malé písmena určujú polohu krabíc
X	Veľké písmena určujú polohu cieľov

Obrázok 25: Vyjadruje reprezentáciu objektov bludiska v textovom súbore

Bludisko je postupnosťou týchto znakov v textovom súbore tak, ako sú evidované v mriežke reálneho bludiska. Príklad bludiska je na obrázku 26.



Obrázok 26: Zobrazuje pôvodné bludisko hry SOKOBAN a jeho SokobanSolver kompatibilnú verziu

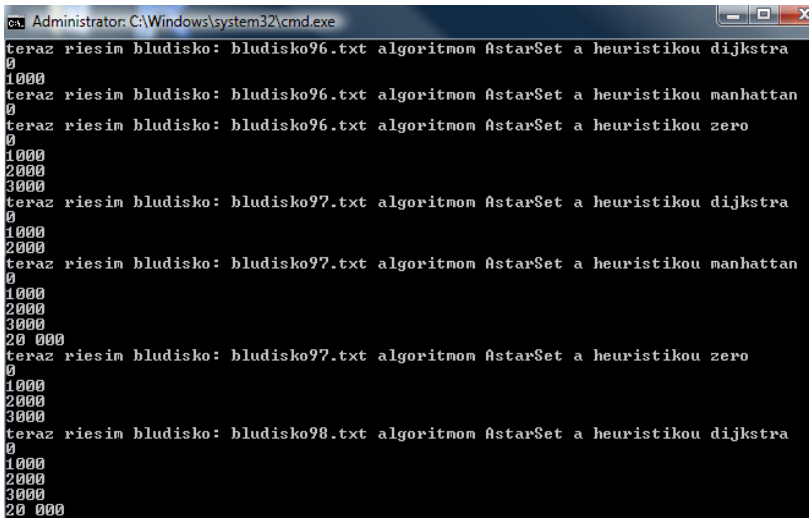
Je nutné, aby boli všetky otvárané vstupné súbory kontrolované na vstupnú kompatibilitu. To znamená že, ak súbor obsahuje znaky rôzne od povolených znakov, program bude bludisko ignorovať a na výstupe konzoly o tom informuje užívateľa.

Výpis na konzolu

Počas behu programu je užívateľ informovaný o priebehu výpočtu. Na displej konzoly program posiela správy o aktuálne riešenom bludisku, o použitom algoritme a heuristike. Uvádza taktiež počet preskúmaných stavov. Počet preskúmaných stavov program kvôli čitateľnosti uvádza iba každých 1000 stavov. Správa na výstupe je formátu:

„teraz riešim bludisko *názov súboru* algoritmom *názov algoritmu* a heuristikou *názov heuristiky"*.

Údaje v tvare písma „*Italic*" sú premenné ktoré budú doplnené, podľa získaných údajov. Príklad výpisu uvádzam na obrázku 27.



Obrázok 27: Ukážka konzolového výstupu programu

Výpis do súboru

Súbory výstupu sú v závislosti od požiadavok užívateľa buď 2 alebo 1. Rozlišujeme súbor určený na výpis počtu krokov získaných z výstupu algoritmov prehľadávania a súbor potrebných časov na vyriešenie bludísk, získaný meraním časových intervalov medzi vstupom do funkcie algoritmu, a výstupom z nej. Výstup v súboroch sa po skončení programu nachádza v textovom formáte. Získané údaje sú do súboru

vypisované priebežne tak ako ich program nadobúda. Každý súbor určený pre výpis má hlavičku:

„bludisko, *názov algoritmu-heuristika*, *názov algoritmu-heuristika*,...“

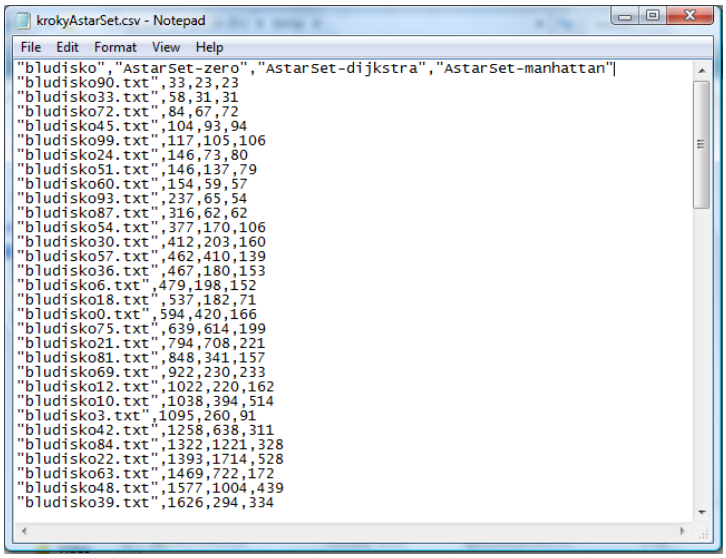
Každý ďalší riadok zápisu do súboru počtu krokov vyzerá nasledovne:

„*názov bludiska*, *počet krokov prvým algoritmom*, *počet krokov druhým algoritmom*,...“

Každý ďalší riadok zápisu do súboru dosiahnutých časov vyzerá nasledovne:

„*názov bludiska*, *čas získaný prvým algoritmom*, *čas získaný druhým algoritmom*,...“

V každom riadku sú údaje získané všetkými algoritmami a heuristikami, zodpovedajúce bludisku uvedenom na počiatku riadku. Na konci výpočtu programu sú údaje v súboroch sformátované do podoby na obrázku 28.



Obrázok 28: Príklad podoby výstupného súboru programu SokobanSolver, v textovom formáte

6.2.2 Programátorský manuál

Dátové štruktúry

Na efektivitu programu vplývajú základné dve používané štruktúry. Už sme o nich hovorili v časti venovanej rozboru informovaného prehľadávania. Sú to zoznamy OPEN a CLOSED. Pripomeniem, že zoznam OPEN slúži na uchovávanie uzlov, ktoré boli

vytvorené pri expandovaní nejakého uzlu a čakajú na preskúmanie. Informované algoritmy si vyberajú stavy na expandovanie podľa vyhodnocovacej funkcie, a to tak, že prednosť má vrchol s najnižšou hodnotou. Je zrejmé že na zoznam OPEN sú kladené nároky vkladanie uzlu a vyberanie minima. Výhodným krokom je teda, vybrať si metódu, ktorá zvláda tieto operácie najlepšie. Z knižnice som si teda vybral kontajner „<multiset>", keďže tento udržiava obsiahnutú postupnosť utriedenú, čím umožňuje vyberať minimum v konštantnom čase. Na porovnanie som implementoval aj vlastnú verziu haldy, pretože tá tiež sľubuje dobrú zložitosť vyberania minima. Navyše halda má dobré známu zložitosť vkladania prvku, a pri tom zachovania usporiadania na svojich prvkoch. Dokáže to s iba logaritmickou zložitosťou. Haldu som implemetoval ako parametrickú triedu. Sama obsahuje ďalší kontajner. Do tohto kontajnera budeme prvky vkladať na koniec. Trieda halda realizuje 5 metód, súvisiacich so správou svojho obsahu. Tri z nich, *erase, emty* a *operátor []* nebudeme ďalej rozoberať. Tieto funkcie iba presúvajú funkčnosti obdobných metód vnútorného kontajnera na povrch triedy. Čo nás zaujíma je metóda insert. Ako to u haldy býva zvykom, táto metóda má za úlohu simulovať vkladanie do binárneho stromu. Prvok sa vloží na poslednú hladinu stromu celkom doľava(na koniec jednorozmerného poľa). Porovnáva svoju hodnotu s otcom a v prípade potreby sa s ním vymení. Tento postup prebieha do chvíle, než sa hodnoty na uzloch ustália. Výsledkom je usporiadaný strom s minimom v koreni. Ak to prenesieme do reči jednorozmerného poľa, tak otec na pozícii s indexom *i*, hľadá svoje deti na pozíciach s indexmi *2i* a *2i+1*. Táto jednoduchá matematika sa prenáša aj do situácie, ktorú využívame pri vkladaní. Teda, deti hľadajú svojho otca na pozícii *(i-1)/2*.

Druhou štruktúrou je zoznam CLOSED. V tomto zozname potrebujeme hlavne hľadať stavy a porovnávať ich navzájom. Keďže priamo na stavoch nieje definované žiadne usporiadanie, sofistikované riešenia nám v tomto prípade nepomôžu. Implementujem teda zoznam CLOSED kontajerom s vkladaním na koniec.

Algoritmy

Program SokobanSolver implementuje 3 algoritmy. Sú to backtracking, Astar, IDAstar. O týchto algoritmoch sme si už čosi povedali v predošlých kapitolách. Vieme aké sú princípy ich fungovania a vieme aj čosi o ich užitočnosti. Teraz sa však pozrieme ako sú naimplementované v praxi, priamo v programe SokobanSolver a na záver práce sa

pozrieme aj na nejaké výsledky behu týchto algoritmov.

Prvým implementovaným bol algoritmus **backtracking**. Neinformované prehľadávanie do hĺbky. Hlavná myšlienka behu tohto programu je obsiahnutá vo funkcii *move()*. Funkcia *backtrack()* vytvorí zoznam šablón, aplikovaním ktorých na aktuálny stav dostaneme všetky možné stavy vychádzajúce z toho stavu, dĺžky jeden krok. Ešte v tejto funkcii, po skontrolovaní či sa náhodou nejedná o cieľový stav, vygenerujeme prvého následníka prvého vrcholu. Toto generovanie má nAstarosti rekurzívna funkcia *move()*. Tá sa po vygenerovaní pozrie či vytvorený stav nie je cieľovým, a ak nie, tak vytvorí prvého následníka tohto vrcholu zavolaním funkcie *move()*. Ak dôjdem do stavu, ktorý nemá už žiadnych ďalších následníkov, ukončím činnosť inštancie funkcie *move(),* na hladine na ktorej sa nachádzam, a takto sa vrátim o hladinu vyššie, na miesto odkiaľ som sa zanáral hlbšie. Za týmto miestom nasleduje buď ďalšie volanie funkcie *move(),* ktoré vygeneruje iného brata, alebo už som vygeneroval všetkých a funkcia move*()* aj na tejto hladine končí. Čím sa opäť dostávame k už prebranému prípadu. Program pokračuje do chvíle, než nájdem riešenie(ak toto riešenie existuje), alebo ukončením funkcie *move*() na hladine dva sa dostanem do funkcie *backtrack*(), ukončenie ktorej signalizuje, že riešenie neexistuje. Musím však upozorniť čitateľa, že proti tomuto postupu je ten použitý v SokobanSolveri trošku odlišný. Z praktických dôvodov, som počet krokov algoritmu obmedzil, istou hornou hranicou. O tejto úprave bude reč na konci kapitoly.

Astar. Algoritmus *Astar* som implementoval dva krát. Raz s použitím dátovej štruktúry set, a raz kde sa zoznam OPEN realizuje prostredníctvom haldy. Princíp fungovania oboch inštancií je však totožný, nakoľko obe štruktúry poskytujú rovnaké nástroje na spravovanie svojho vnútorného obsahu. V tomto texte sa teda bližšie pozrieme iba na jednu inštanciu, pričom všetko čo si o nej povieme je možné analogicky preniesť na druhý zmieňovaný prípad.

Na prezentáciu som si vybral *Astar,* využívajúci štruktúru set. Celý algoritmus je naprogramovaný v jedinej funkcii, ktorá sa nazýva *astar_set()*. Na začiatku pripravíme zoznamy *open* a *closed*. Ako prvý vrchol vložíme do zoznamu otvorených vrcholov aktuálny vrchol (počiatok). Teraz sme pripravený na to, aby sme rozbehli hľadanie. Hľadanie prebieha v cykle. Kontroluje sa podmienka prázdnosti zoznamu OPEN a

podmienka podobná tej o ktorej sme hovorili už pri algoritme backtrackingu. Druhá podmienka ukončuje hľadanie pri presiahnutí určitého počtu prehľadaných stavov.

Rozoberme si teda ako vyzerá jeden prechod cyklom. V prvom rade si vyberieme stav s najmenšou hodnotou. Na uchovávanie používame *set*, takže tento stav je prvý v poradí. Funkciou *is_final()* si overíme či tento stav nieje náhodou našim hľadaným stavom. Funkcia *is_final()* dostane na vstup, štruktúru reprezentujúcu aktuálny stav a zoznam cieľových polôh krabíc a vráti *true* v prípade, že každá krabica má polohu identickú s príslušnou cieľovou pozíciou. Funkcia si volá na pomoc preťaženú funkciu *same()*, ktorá v jednom prípade overuje postavenie krabíc voči cieľovým pozíciám a pritom ignoruje postavenie hráča, v druhom berie do úvahy aj polohu hráča, čo jej umožňuje podávať správu o odlišnosti stavov. V prípade že *is_final()* uzná stav za konečný, beh funkcie sa ukončí, ďalej sa neprehľadáva a na výstup sa pošle veľkosť zoznamu *closed*. Táto hodnota reprezentuje počet prehľadaných a uzavretých vrcholov. Je práve tou hodnotou o ktorej sme hovorili v úvode a ktorá nám pomôže viesť štatistiku úspešnosti algoritmu. Ak nás nevyhodila z cyklu táto funkcia je zrejmé, že tento prechod už prejdeme do konca. To znamená, že na tomto mieste môžeme presunúť vrchol zo zoznamu *open,* do zoznamu *closed.* Vymazať prvý prvok v *open*, vložiť ho kamkoľvek do *closed*(v našom prípade na koniec). Sme pripravený expandovať vrchol a generovať tak jeho 4 následníkov. Stavy vytvára funkcia *create_moves()*. Nasleduje funkcia *correct_move()*, ktorá overuje korektnosť krokov a prepustí len tie, ktoré majú podľa pravidiel hry zmysel. Ďalšou funkciou, ktorá môže skresať celkový počet krokov ktorými sa budeme v budúcnosti zaoberať je funkcia *contains()*. Jej úlohou je overiť si jedinečnosť stavu v porovnaní so zoznamom *closed*. Týmto sa zbavíme stavov, ktoré sme už uzavreli, teda už poznáme najlepšiu cestu do nich. Posledný hráč, ktorý má možnosť upraviť počet stavov, ktoré posunieme medzi tie ktorými má zmysel sa zaoberať, je opäť funkcia *contains()*, avšak tentokrát kontroluje stav proti zoznamu *open*. V prípade, že sme už tento stav v zozname *open* mali, overíme si, či má ten obsiahnutý v zozname nižšiu hodnotu cesty z počiatku do stavu. Ak áno, na nový zabudneme. Ak nie, tieto stavy vymeníme a novému stavu dopočítame aj ostatné hodnoty. V prípade, že sa v zozname tento stav nenachádza vôbec, dopočítame mu ostatné hodnoty a stav zaradíme do zoznamu open. Skrátene, tento postup vyberie najlepší stav, overí si že nie je konečný, expanduje ho, preverí jeho potomkov na

korektnosť a zaradí ho do zoznamu typu *OPEN*. Tento postup nazývame prebranie stavu. Ak máme stav prebraný, môžeme sa pustiť do preberania ďalšieho, kým nenájdeme cieľ, v prípade, že existuje, alebo vyprázdnime zoznam *open*, v prípade, že cieľ neexistuje. Jeden či druhý prípad ukončuje funkciu *astar_set()*, program zapíše zistené údaje do súboru, a presunie sa k ďalšiemu algoritmu.

Týmto algoritmom je algoritmus **IDAstar**. Naprogramovaný je prostredníctvom dvoch funkcií. Vzhľadom k tomu, že funguje na princípe prehľadávania do hĺbky, rozhodol som sa pre implementáciu rekurziou. Rekurzívna je funkcia *ida_search()*. Ak však pôjdeme po poradí pri zahájení hľadania vstúpime do funkcie *ida_star()*. V tejto funkcii zadefinujeme hodnotu *treshold*, čo je medza ponoru, určená pre začiatok heuristickým odhadom vzdialenosti počiatku a cieľa. Zadefinujeme počiatok, a zoznam kam budeme dávať použité stavy. Počiatočný stav vložíme do zoznamu použitých stavov a spustíme hľadanie pre súčasnú hodnotu *treshold*, spustíme teda funkciu *ida_search()* na počiatočný vrchol. V úvode funkcie overíme, či sa náhodou nejedná o cieľový stav a odhadneme vzdialenosť z vrcholu do cieľa. Použijeme tento údaj na výpočet celkovej odhadovanej dĺžky cesty a porovnáme ho s medzou. Ak je cesta dlhšia než je medza hlboká, ukončíme funkciu *ida_search()*. Dostaneme sa tak na vyššiu hladinu, na miesto odkiaľ sme volali *ida_search()*. Ak je touto hladinou funkcia *ida_star()*, zvyšujeme hodnotu medze a opäť sa ponárame do rekurzie. Ak je ňou iná inštancia funkcie *ida_search()*, vytvorí sa ďalší následník aktuálne skúmaného vrcholu a opäť sa zanoríme do rekurzie. V prípade, že vrchol vyhovuje medzi, uznáme, že má zmysel sa ním zaoberať a pomocou funkcie *create_moves()* vytvoríme 4 nové stavy. Pre každý z nich, postupne kontrolujeme, či sú zmysluplné a ponárame sa do nich. Ak sme už vyskúšali všetky štyri vrcholy, dorazíme na koniec funkcie a opustíme ju na mieste o úroveň vyššie, odkiaľ sme ju pôvodne zavolali. Program od tohto bodu už pokračuje jedným z vyššie popísaných spôsobov. Postup sa opakuje do chvíle než algoritmus nájde riešenie, ak existuje, alebo bude ukončený ako všetky ostatné algoritmy hornou medzou počtu prebratých vrcholov.

Heuristiky

V programe sú implementované dve heuristiky. Jednou z nich je heuristika Manhattan o ktorej už tejto práci bola reč. V SokobanSolveri je táto heuristika implementovaná

funkciou *manhattan()*. Po zadaní hracieho plánu, zoznamu krabíc a zoznamu cieľových stavov, vypočíta postupne vzdialenosti všetkých krabíc od cieľových stavov a spraví ich sumu, ktorú pošle na výstup. Vzdialenosť krabice a cieľového stavu je určená ako absolútna hodnota rozdielu x-ových súradníc krabice a cieľa, pripočítaná k absolútnej hodnote rozdielov y-ových súradníc krabice a cieľa.

Druhou zaujímavejšou heuristikou je heuristika implementovaná vo funkcii *estimated_distance()*. Táto heuristika je založená na určení vzdialenosti predmetov na pláne, na základe ich určenia najkratších vzájomných vzdialeností pomocou Dijkstrovho algoritmu. Algoritmus je implementovaný vo funkcii *dijkstra()*. Tento algoritmus využíva dátovú štruktúru *distances* v ktorej sú uložené jednotlivé vzdialenosti po tom, čo už boli niekedy vyhľadané. Na začiatku funkcie sa teda pozrieme, či sme už vzdialenosť medzi vrcholmi niekedy nepočítali. Ak áno, ihneď ju vrátime a ďalej sa hľadaním nezaoberáme. Pre potreby algoritmu si vytvoríme pole nesúce informáciu o každom bode plánu, či bol alebo nebol dosiahnutý. Na počiatku nebol dosiahnutý žiaden, preto všetky hodnoty dostanú *false*. Ďalšou štruktúrou ktorá ponesie podporné informácie je pole E-hodnôt, pre každé pole plánu. Poslednou je riadiaca štruktúra *reached*. V nej sú uložené jednotlivé polia. Na začiatok tam pridáme vrchol z ktorého štartujeme hľadanie. Môžeme začať hľadať. Hľadanie prebieha v cykle, kde sa v každom kroku vyberie pole s najmenšou E-hodnotou. Zoznam *reached* je implementovaný ako *multiset*, takže toto pole sa nájde jednoducho. Je to prvý prvok zoznamu. V prvok kroku teda vezmeme z *reached* prvé pole, jeho E-hodnota je od začiatku implicitne definovaná ako 0. Touto hodnotou vyplníme príslušné polie v tabuľke *distances*. Teda to pole tabuľky *distances*, ktoré v riadku aj v stĺpci označuje práve preberaný vrchol. Výsledok prvého kroku je teda nulová vzdialenosť medzi počiatkom hľadania a prvým vrcholom, čo je počiatok hľadania. V ďalšom kroku sa nachádza cyklus, ktorý pre každého bezprostredného suseda, na ktorého sa dá vstúpiť, vypočíta E-hodnotu vzhľadom k E-hodnote vrcholu, z ktorého boli označení ako susedia. Samozrejme si overujeme, či táto hodnota už nieje známa. V tom prípade sa presúvame na ďalšieho suseda. Následne overujeme, či dočasná E-hodnota zistená pri tejto vlne hľadania nie je väčšia, ako hodnota ktorú sa chystáme poľu priradiť, ak sme teda pri tomto hľadaní už pole navštívili. Ak je nová hodnota menšia, označíme pole za navštívené(je možné, že už nesie hodnotu o navštívení), pridáme vrchol do zoznamu

reached a nastavíme jeho E hodnotu na novú hodnotu. Iterácia vnútorného cyklu končí, keď takto spracujeme všetkých susedov aktuálneho poľa a potom sa vonkajším cyklom presunieme na ďalší vrchol zo zoznamu. Určite sme si už všimli, že sa jedná vlastne prehľadávanie do šírky. Pri každom prechode funkciou, nájdeme vzdialenosti všetkých dostupných polí od počiatočného poľa. Nakoniec vrátime vzdialenosť vrcholov *u* a *v,* o ktoré nám šlo v prvom rade. Táto metóda je výpočtovo zložitejšia ako Manhattan, ale na druhú stranu sľubuje lepší odhad vzdialenosti.

Týmto sme ukončili analýzu kľúčových častí programu. Teraz, keď máme dostatočné teoretické základy algoritmov a poznáme aj ich implementáciu v programe SokobanSolver, pozrime sa na už na konkrétne výsledky ich práce.

6.3 Testovanie

Jednou z úloh tejto práce bolo aj preskúmať konkrétne výsledky činnosti študovaných algoritmov. V následujúcich riadkoch sa pozrieme na to, ako sme jednotlivé bludiská testovali. Na výsledky získané popísaným testom sa pozrieme na konci práce.

6.3.1 Popis testu

Testovanie prebiehalo programom *SokobanSolver* na bludiskách vytvorených programom *SokobanGenerator*. Bludiská sú generované s parametrami, o ktorých som už písal v kapitole venovanej SokobanGenerátoru. Testovali sa 3 algoritmy. Sú to Astar, IDAstar a *backtracking*. Testovanie na algoritme Astar prebiehalo v dvoch formách, jedná z nich používa ako hlavnú pamäťovú štruktúru množinu *set(astarSet)* a druhá používa haldu(*astarHeap*). Testovalo sa 500 vygenerovaných bludísk na ktorých boli postupne preskúšané všetky prípustné dvojice algoritmus-heuristika.

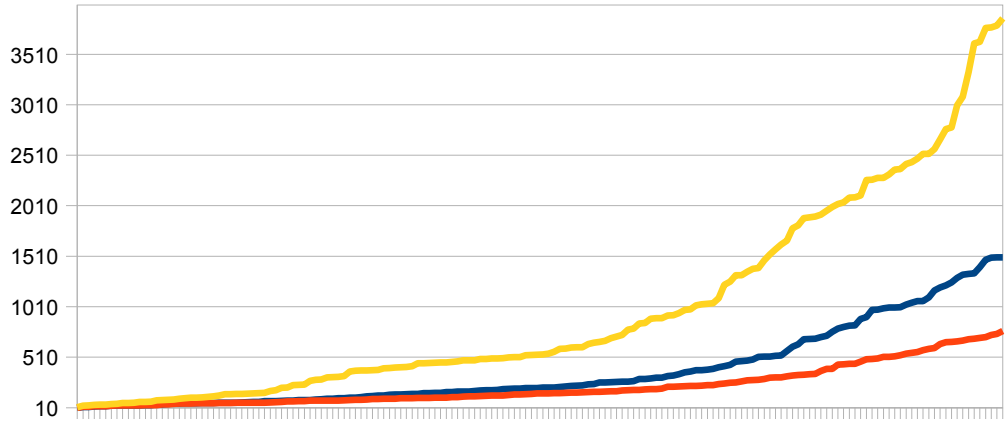
Ako vieme *SOKOBAN* je NP-ťažký a PSPACE úplný problém. Očakávania na pozitívne výsledky preto smerujú hlavne k informovaným algoritmom. IDAstar sa podľa teórie ukazuje byť asymptoticky rovnako efektívny ako Astar, takže výsledky by mali byť porovnateľné. Boj medzi algoritmami Astar využívajúcich rôzne dátové štruktúry by mal mať vplyv iba na časový údaj.

Čo sa týka heuristiky, tak *Dijkstra* má v niektorých konkrétnych prípadoch viac sily, ale na druhú stranu je výpočetne zložitejšia. *Manhattan,* je podľa odhadu, pre tento typ

bludiska lepším algoritmom vďaka svojej časovej jednoduchosti a pomerne dobrému odhadu.

6.3.2 Výsledok testu

Prvým testovacím algoritmom bol algoritmus Astar na dátovej štruktúre set. Nasledujúci graf nám ukáže, ako sa pri použití s týmto algoritmom chovali naprogramované heuristiky. Priložíme aj údaje pre algoritmus Astar bežiaci bez heuristiky.

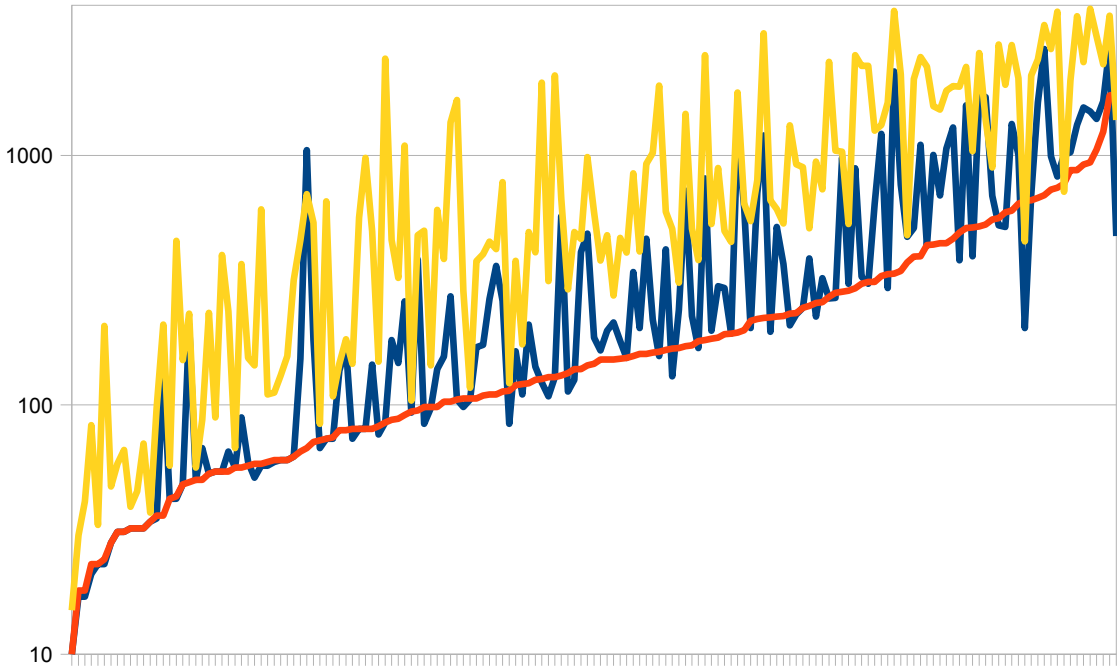


Graf 31: Graf zobrazuje ztriedené hodnoty počtu krokov pre jednotlivéh heuristiky v splupráci s algoritmom AstarSet. Modrá farba reprezentuje heuristiku dijkstra, červená manhattan a žltá je bez heuristiky.

Ak si pozrieme graf triedených hodnôt počtu krokov behu algoritmu Astar, zistíme, že najhoršie dopadla heuristika zero. To nás neprekvapí. Čo je však zaujímavejšie je, že heuristika manhattan dopadla v tomto teste najlepšie, napriek tomu, že je omnoho primitívnejšia než heuristika dijkstra. Dalo by sa očakávať, že heuristika manhattan bude taká dobrá alebo horšia ako heurstika dijkstra. Výsledky testu však hovoria, že je to presne naopak.

Pozrime sa teraz na ďalší obrázok na ktorom sú hodnoty triedené podľa jednej vedúcej tak, že zachovávajú príslušnosť k jednotlivým bludiskám. Tento graf nám umožní sa pozrieť na konkrétne bludiská a pochopiť, prečo sa algoritmus správal tak, ako sa

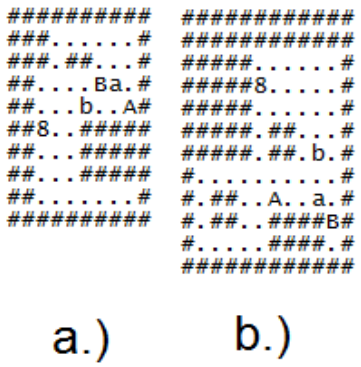
správal.



Graf 32: Počet krokov algoritmu Astar. Červená čiara reprezentuje heuristiku manhattan, modrá čiara heuristiku dijkstra, a žlta čiara je Astar bez heuristiky

Graf 32 ukazuje, že napriek jasnej prevahe heuristiky manhattan sa občas stane, že ju dijkstra predbehne. Dokonca aj Astar bez heuristiky má v niektorých prípadoch lepší výsledok.

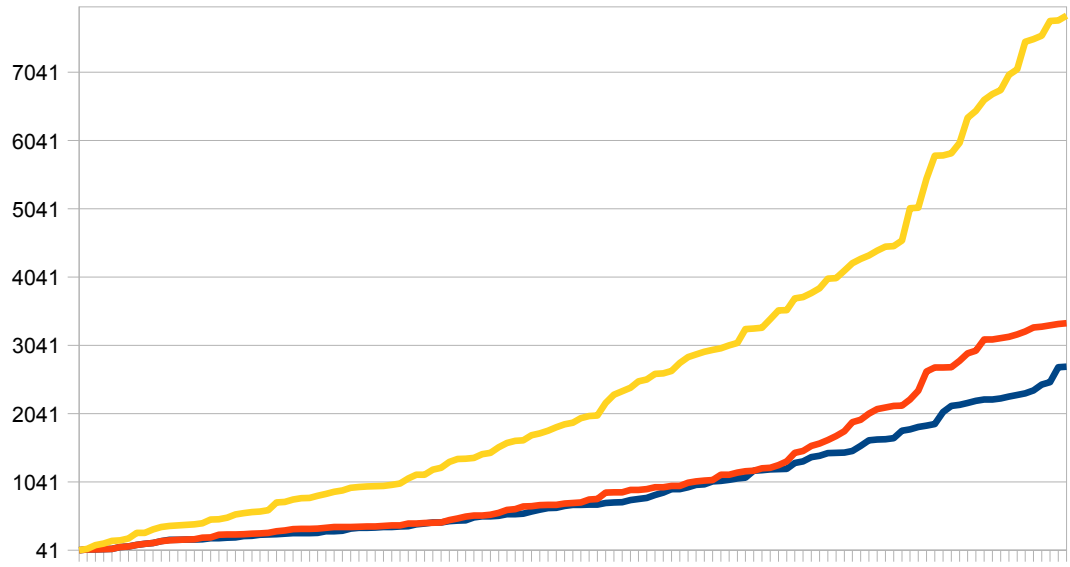
Pozrime sa teda, ako vyzerá jeden súbor na ktorom vyhráva manhattan a porovnajme ho so súborom, na ktorom vyhráva dijkstra s Astarom bez heuristiky.



Obrázok 29: na obrázku vľavo je schéma bludiska na ktorom manhattan zlyhal, na obrázku b.) bludisko kde dobehol s najväčším rozdielom počtu krokov proti zvyšným algoritmom.

Na obrázku a.) je súbor v ktorom heuristika dijsktra bola 3-krát rýchlejšia než manhattan a 2-krát rýchlejšia než Astar bez heuristiky. Tranzitívne, Astar bez heuristiky bol rýchlejší než manhattan. Na obrázku b.) je zas bludisko, v ktorom manhattan najviac „ušiel“ konkurencii.

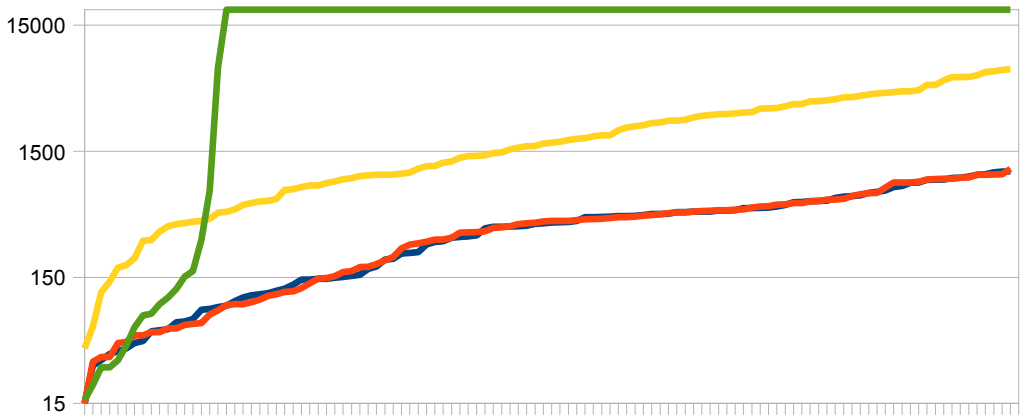
Pred tým než sa pozrieme na porovnanie jednotlivých algoritmov, pozrime sa ešte na graf, porovnávajúci výsledky behu heuristík s iným algoritmom. Týmto algoritmom bude algoritmus IDAstar.



Graf 33: Počet potrebných krokov pre heuristiky spolupracujúce s IDAstar algoritmom. Modrá je manhattan, červená dijkstra, žltá IDAstar bez heuristiky

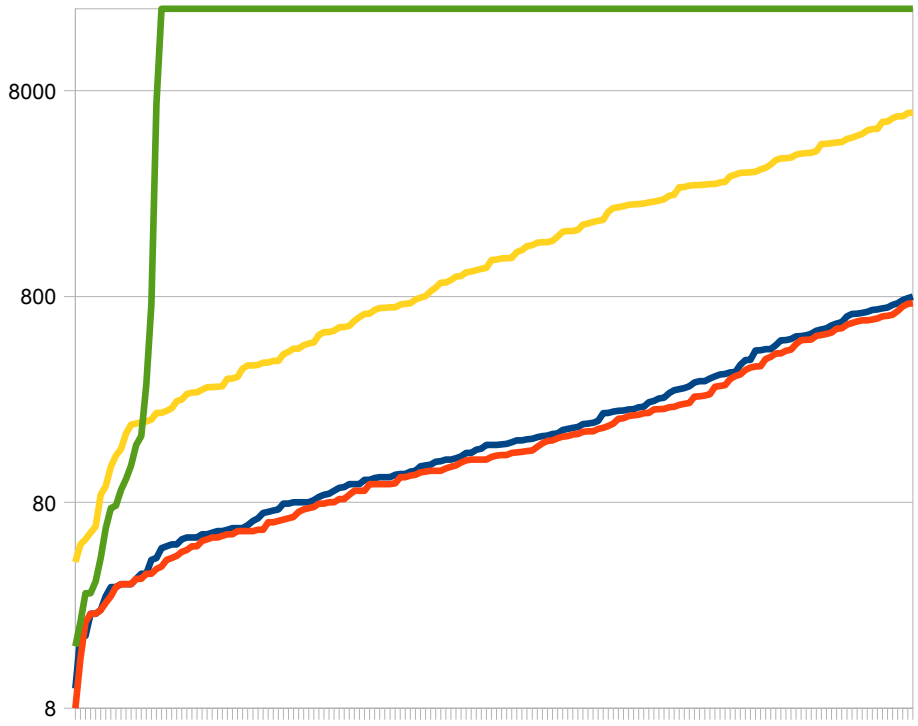
Z obrázku možno vidieť, že principiálne sa proti algoritmu Astar nič nezmenilo. Rozdiel medzi dijkstrou a manhattanom sa síce zdá byť menší, ako v predošlom prípade, avšak to je spôsobené väčším rozsahom krokov na ose y.

Poďme sa teraz pozrieť na porovnanie jednotlivých algoritmov. Najprv porovnáme informované algoritmy bez heuristiky s algoritmom backrackingu.



Graf 34: Porovnanie výkonnosti algoritmov, informované algoritmy majú nastavenú heuristiku 0. Aglritmy: zelená-backtracking,žltá-IdaStar,červená-AstarHeap,modrá-AstarSet

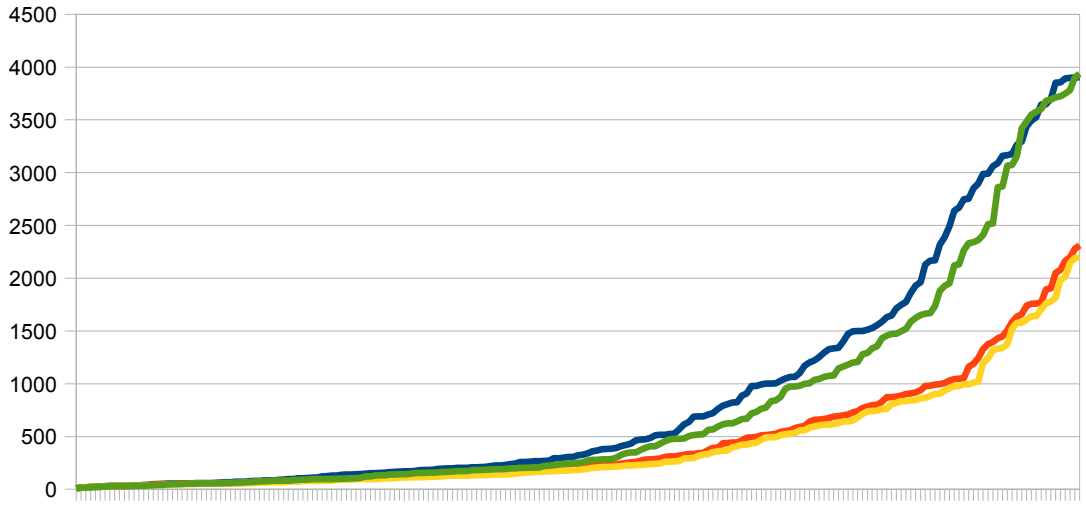
Graf 34 jasne hovorí v neprospech algoritmu backtracking. Jeho správanie by sme mohli označiť ako „dychtivé“ v zmysle, že ak algoritmus nájde riešenie, tak to spraví skutočne rýchlo. Ak je však problém zložitejší, je veľká pravdepodobnosť, že sa výsledku nedočkáme vôbec. Napriek tomu, že tento výsledok nemôže nikoho prekvapiť, jeden z algoritmov, výrazne sklamal očakávania. Jedná sa o algoritmus IDAstar. Pracoval v rovnakých podmienkach ako oba Astary, bez heuristiky, napriek tomu citeľne zaostáva. Ďalším prekvapením je, že oba Astary napriek veľkej podobnosti nemajú rovnaký výstup. Na tento jav sa však pozrieme neskôr. Teraz nás čaká porovnanie algoritmov s nasadenou najúčinnejšou naprogramovanou heuristikou manhattan, proti heuristike backtracking. Dopredu neočakávame výraznejšie zmeny proti grafu 34, no bude zaujímavé sledovať, nakoľko sa polepší algoritmus IDAstar. Pozrime sa teda na graf 35.



Graf 35: Reprezentácia počtu krokov algoritmu, s použím heuristiky manhattan. Zelená-backtracking, žltá-IDAstar, modrá-AstarSet, červená AstarHeap

Výsledkom použitia heuristiky manhattan je ešte výraznejší rozdiel medzi informovanými algoritmami a backtrackingom. Na druhú stranu, rozdiel vo výkonnosti IDAstar proti Astar ostal zachovaný. Oba algoritmy sa posunuli nižšie, smerom k ose x.

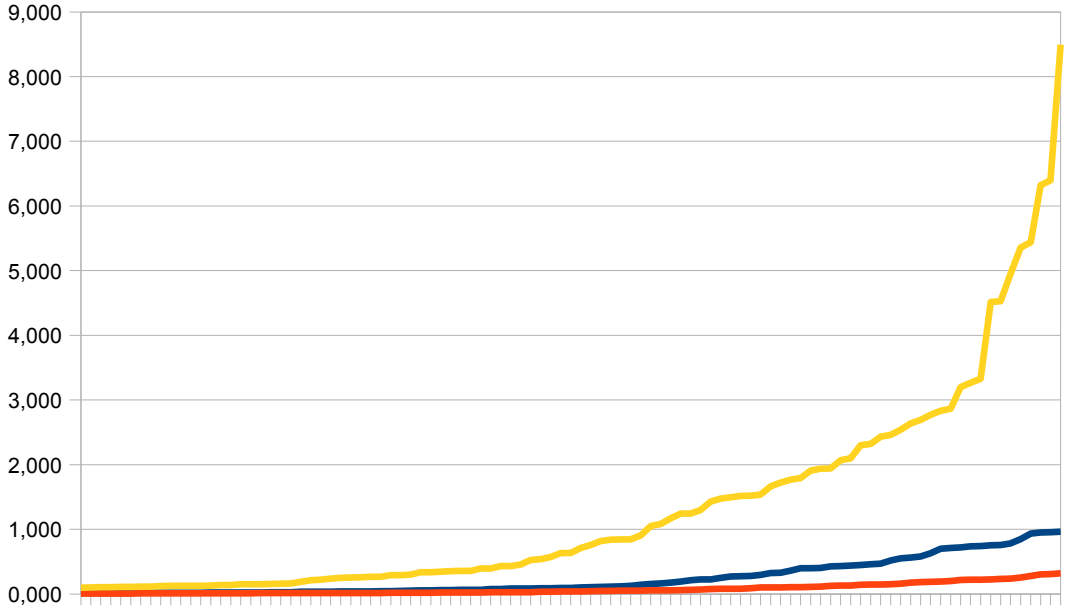
Teraz keď už sme sa pozreli na výkonnosti algoritmov a heuristík ostáva nám ešte porovnať dopad na výkonnosť hľadania použitie dátovej štruktúry set proti použitiu haldy. Na tento test použijeme algoritmus Astar.



Graf 36: Reprezentuje početk krokov pre: modrý-AstarSet-dijkstra, zelený-AstarHeap-dijkstra,červený AstarSet-manhattan,žltý AstarHeap-manhattan algoritmus.

Graf 36 nám ešte viac odhalil zaujímavú vlastnosť štruktúr haldy a množiny. V oboch prípadoch vyhráva množina. Množina, ako aj halda, udržiavajú utriedenú postupnosť z ktorej sa vyberá vždy prvý prvok. Ako je potom možné, že jedna má lepšie vlastnosti než druhá? Jedinou možnosťou je, že obe tieto štruktúry sa líšia tým, ako nakladajú s pridávanými prvkami, ktoré majú rovnakú hodnotu ako minimum. Štruktúra haldy naprogramovaná v programe SokobanSolver netlačí nový prvok až na úplný začiatok. Zato množina set áno. Druhý menovaný postup sa ukazuje byť, aj keď len nepatrne, no predsa výkonnejší.

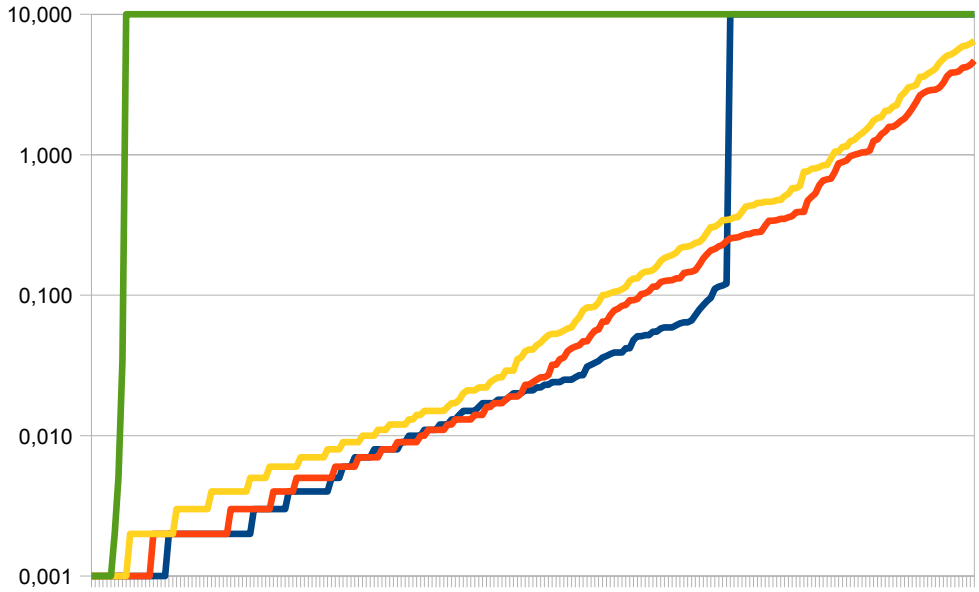
Menší počet prehľadávaných krokov nutne nemusí znamenať automaticky rýchlejší algoritmus. Výkonnosť algoritmu závisí aj od výpočtovej zložitosti prehľadávania jedného kroku, ktorá závisí priamo aj od zložitosti výpočtu heuristickej funkcie. Pozrime sa na záver teda aj na porovnania dĺžky trvania behu jednotlivých algoritmu Astar s rôznymi heuristikami. Túto situáciu popisuje graf 37.



Graf 37: Zobrazenie dĺžky behu algoritmu Astar s použitím heuristiky: červená manhattan, modrá-dijkstra, žltá-bez heuristiky

Graf ilustruje predpokladaný výsledok. Funkcia dijkstrovej heuristiky je výpočtovo zložitejšia než funkcia heuristiky manhattan. V predošlých odstavcoch sme navyše zistili, že heuristika manhattan vykazuje lepšie celkové výsledky než heuristika dijkstra.

Posledným porovnaním bude celkové porovnanie dĺžky behu algoritmov s použitím heuristiky manhattan v prípade informovaných algoritmov. Tieto výsledky ilustruje nasledujúci graf.



Graf 38: Reprezentácia dĺžky behu jednotlivých algoritmov: zelený backtracking, žltý AstarSet, červený AstarHeap, modrý IDAstar

Graf 38 potvrdzuje známe zistenie z predošlých meraní. A to síce, že algoritmus backtracking ak nájde riešenie, tak sa tak stane veľmi rýchlo, avšak riešenie nájde iba pre najjednoduchšie problémy. Algoritmus IDAstar sa ukazuje výpočtovo rýchlejší pre jednoduché problémy než algoritmus Astar, no pomerne skoro mu dochádza dych a jeho hodnota dĺžky času potrebného na vyriešenie problému začína prudko stúpať. Tento fakt iba potvrdzuje to, čo sa nám podarilo zistiť o algoritme IDAstar v predošlých meraniach. Tak isto sme si potvrdili mierne lepšiu výpočetnú zložitosť algoritmu Astar s použitím pamäťovej štruktúry haldy.

7 Záver

Cieľom práce, ako sme si ho definovali v úvode, bolo vytvoriť komplexného sprievodcu problematikou prehľadávania stavového priestoru, aplikovanou na príklade hry SOKOBAN. Krok po kroku sme sa dostali cez definíciu problému, ktorú sme si popísali jazykom STRIPS, k charakteristikám jeho zložitosti. Ukázali sme si, že SOKOBAN je NP-ťažký a PSPACE-úplný. Došli sme až k metódam jeho riešenia. Tu sme si predstavili informované aj neinformované algoritmy prehľadávania stavového priestoru, vyslovili sme množstvo tvrdení o vlastnostiach týchto algoritmov. Zamerali sme sa aj na formálny popis funkcií heuristík, ktorých použitie je kritické pri takto obtiažných úlohách. Postupne sme si takto vybudovali silné povedomie o povahe študovaného problému a možnostiach jeho riešeniach. V praktickej časti sme si povedali niečo o programe SokobanSolver, ktorý využíva predstavené algoritmy a heuristiky. Ukázali sme si ako sa tento program používa a predstavili sme si aj kľúčové zložky tohto programu z hľadiska programátora. Prestavili sme si tiež program SokobanGenerator, generujúci automaticky veľké množstvo bludísk. Tými to bludiskami sme následne naplnili program SokobanSolver a v poslednej časti práce sa pozreli na jeho výstupy. Výsledkom týchto pozorovaní sú nasledujúce postrehy:

1. Prvý prekvapujúci záver vyplynul z výsledkov získaných heuristikou dijkstra v porovnaní s heuristikou manhattan. V dvoch testoch používajúcich rozdielne algoritmy sa manhattan stal jasne výkonnejšou heuristikou. Navyše výpočet heuristiky manhattan je neporovnateľne rýchlejší než algoritmus dijkstra. Z čoho vyplýva, že manhattan sa stal víťazom testu heuristík.
2. Druhým zaujímavým záverom je nie veľmi uspokojivá výkonnosť algoritmu IDAstar. Proti neinformovanému algoritmu je IDAstar stále ďaleko výkonnejší, no v porovnaní s algoritmom Astar v oboch prevedeniach citeľne zaostáva.
3. Voľba dátových štruktúr má vplyv na použité algoritmy. Porovnanie testu algoritmu Astar v prevedení set a v prevedení heap ukázal rozdiel, aj keď nie významný, v ich výkonnosti. Tento jav prikladám rozdielnemu nakladaniu s v kladnými hodnotami rovnajúcimi sa minimu.

Pôvodný zámer vylepšenia heuristiky, pochádzajúci zo zadania práce, sa tak skrze

dijkstrovu heuristiku naplniť nepodarilo, avšak aj informácia o nie prílišnej vhodnosti tejto heuristiky môže v budúcnosti pomôcť pri výbere vhodného postupu.

Literatúra

[1]Sokoban University of Alberta, http://www.cs.ualberta.ca/~games/Sokoban/

[2]Sokoban Homepage, http://www.sokoban.jp/

[3] Culberson C.J.(1997): Sokoban is PSPACE-complete,Department of Computer Science, The University of Alberta

[4]Mařík V., Štěpánková O., Lažanský J. a kol.(1993): *Umělá inteligence(3)*, Academia.

[5]Freebase:http://www.freebase.com/view/en/pspace-complete

[6]Wolfgang R., Grzegorz R.: Basic Models, Springer

[7]Dorit D., Uri Z.(1995): Sokoban and other motion planning problems

[8] Demaine D. E., Hoffmann M.: *Pushing Blocks is NP-Complete for Noncrossing Solution Paths*

[9] Garey M., Johnson D.(1979): Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York, NY,

[10]Russel S., Norvig P.(2003): *Artificial Intelligence A Modern Approach Second Edition*, Prentice Hall.

[11]Küngas P.: Logic and AI Planning: STRIPS, Department of Computer and Information Science Norwegian University of Science and Technology

[12] Nilsson J. N.(1971): *Problem-Solving Methods In Artificial Intelligence,*McGraw-Hill Book Company.

[13]Mařík V., Štěpánková O., Lažanský J. a kol.(1993): *Umělá inteligence(1)*, Academia.

[14]Murase Y., Matsubara H., Hiraga Y.: *Automatic Making of Sokoban Problems*, University of Library and Information Science, Japan.